Universidad
Carlos III de Madrid

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA EN INFORMÁTICA

MOTOR DE *DEFERRED RENDER* PARA XNA

PROYECTO FIN DE CARRERA

Madrid, Diciembre 2013

Autor: Gabriel de la Cruz Fortún

Tutores: Javier García Polo

Vidal Alcázar Saiz

AGRADECIMIENTOS

A mis tutores, por su ayuda y comprensión durante la realización del proyecto.

A mi familia y amigos, por haberme apoyado siempre y animarme a seguir adelante.

A Vidal, por hacer esto posible cuando ni yo lo creía.

RESUMEN

Este proyecto propone una arquitectura alternativa de renderizado para el motor de desarrollo de videojuegos XNA. Esta arquitectura utilizará el modelo de renderizado en diferido (deferred render) en lugar de usar las funciones habituales del motor de XNA. En concreto, se utilizará deferred shading aunque sería sencillo pasar a deferred lighting para utilizar materiales más complejos.

El desarrollo de esta arquitectura se realizará en tres pasos principales. En el primer paso será necesario procesar los modelos utilizados para usar un shader propio que genere los distintos g-buffers. Posteriormente (en diferido) se realizará la fase de iluminación, renderizando cada luz como si fuera un objeto con un shader que toma como entradas la información de los g-buffers y añade a la escena la iluminación aportada por esa fuente de luz. Finalmente, con la iluminación de todas las luces de la escena y la información de color del g-buffer se compone la escena final.

ABSTRACT

This project proposes an alternative rendering architecture for the videogame development engine XNA. This architecture will use the deferred rendering model instead of using the standard functions of the XNA engine. Specifically, deferred shading will be used although it would be easy to change it to deferred lighting to manage complex materials.

Development of this architecture will be done in three main steps. In first step it will be required to process the 3D models to use a custom shader that outputs all the g-buffers. Later, in a deferred stage, lighting will be calculated, rendering each light as an object with a shader that takes the information in the g-buffers as inputs and adds to the scene the light generated by that source. Finally, we compose the final scene with all the lighting of the scene and the color information stored in the g-buffer.

ÍNDICE DE CONTENIDOS

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

INTRODUCCIÓN

VISIÓN GENERAL

Este proyecto consiste en el desarrollo de un sistema de renderizado diferido (deferred render) para el motor de videojuegos XNA.

El renderizado diferido es una técnica para mostrar objetos iluminados en pantalla de una forma alternativa, procesando por separado la geometría y las luces en la escena. Para lograr esto nuestro sistema renderizará primero los objetos a unos buffers de geometría y luego usará estos para procesar individualmente el aporte de cada luz.

El sistema está dividido en dos partes: una librería que procesará los recursos gráficos y la librería principal que proporciona las clases necesarias para cargar y renderizar escenas con un número variable de objetos y luces.

Adicionalmente, el proyecto constará también de un sencillo visor que hará uso de la librería para mostrar distintas escenas de ejemplo.

MOTIVACIÓN

En los últimos años, el mundo de los videojuegos está sufriendo una progresiva transformación. El desarrollo de videojuegos, algo que se había establecido como algo lejano y exclusivo de los estudios profesionales, está viendo como empiezan a surgir otro tipo de videojuegos realizados por equipos de pocas personas, a veces incluso por una sola.

No es algo nuevo, en sus inicios la industria del videojuego creció a partir de una generación de jugadores que tenían acceso a los medios adecuados para crear sus propias aventuras. Pero, a medida que los videojuegos fueron complicándose y los jugadores se volvían más exigentes, el desarrollo de videojuegos empezó a requerir equipos más grandes y una mayor inversión de tiempo y dinero.

De algún modo, se asumió que al ser más elaborados y tener una mejor presentación (mejores gráficos, mejor sonido, etc.) los juegos serían más divertidos. Pero la industria del videojuego es aún muy joven y está en continua evolución. Con el tiempo, el público más exigente ha ido poco a poco cansándose de los modelos clásicos de videojuegos (plataformas, *shooters*, juegos de carreras) que aunque cada vez son más realistas no ofrecen nada nuevo.

Con el alto coste de producción actual de un videojuego son pocas las empresas que se atreven a salirse del camino marcado y aquellas que lo hacen muchas veces pagan un alto precio por ello. Esto, unido a la gran difusión y fácil accesibilidad de las nuevas tecnologías, abre un mundo de posibilidades para aquellas propuestas más creativas que no necesitan competir en valores de producción con el mercado más convencional.

Esta vuelta al desarrollo de garaje es ya un hecho y tiene su propio mercado que coexiste con el de las superproducciones. Tanto es así que está recibiendo el apoyo de las grandes marcas tanto a nivel de visibilidad como a nivel de accesibilidad a las herramientas de desarrollo.

Sin embargo, a menudo los desarrollos de videojuegos independientes se ven limitados por las funcionalidades expuestas por los proveedores de los distintos motores de videojuegos. Este tipo de desarrollos utiliza habitualmente motores que son gratuitos o de bajo coste y que, por lo tanto, no ofrecen los últimos avances tecnológicos. En cierto modo esto impide competir con productos con mayor presupuesto y no necesariamente mayor calidad.

En el caso que nos ocupa el motor XNA de Microsoft ofrece una gran sencillez de uso pero muy poca flexibilidad a la hora de personalizar la forma de presentar los gráficos. Aunque existe la posibilidad de renderizar un objeto usando shaders personalizados (fundamental para conseguir un apartado gráfico no genérico) no es fácil integrarlo con la sencilla funcionalidad de dibujado ofrecida por el motor.

Este proyecto ofrecerá una alternativa de presentación utilizando renderizado en diferido (deferred rendering), más parecida a la utilizada por los motores comerciales actuales. La principal ventaja de esta arquitectura es la de separar la iluminación del dibujado de los objetos. De esta forma será posible utilizar tantas fuentes de luz por objeto como sean necesarias, esquivando así la limitación en el número de luces impuesta por la arquitectura por defecto de XNA. Al contrario de lo que pueda parecer, esto no implica necesariamente un peor rendimiento. Con la arquitectura por defecto, cada pixel de objeto debe calcular la iluminación recibida por cada fuente de luz asociada, aunque ésta no le esté afectando. En cambio, en el renderizado en diferido las luces no están asociadas a ningún objeto si no que afectan al conjunto de la escena, calculando únicamente aquellos pixeles que caen dentro de su rango. Por tanto mientras que antes lo determinante era el número de luces, ahora lo será el tamaño de estas en pantalla (el número de pixeles a los que afectan).

Otra ventaja adicional es que, al estar separadas las luces de los objetos, no será necesario realizar la a veces costosa tarea de relacionar cada objeto con las luces que le deben afectar. Aparte de ahorrarnos la complejidad que supone en escenas con muchas luces, evita también muchos errores que pueden ocurrir cuando una luz que físicamente debería afectar a un objeto no lo hace (ya sea por no estar correctamente asociada o por haberse superado el número de luces).

OBJETIVOS

El objetivo del proyecto es crear una librería para XNA que permita renderizar escenas compuestas por varios objetos y varias luces utilizando la técnica de deferred shading, que no existe en la implementación actual.

Para conseguir este objetivo principal hay que distinguir varios objetivos parciales que habrá que implementar por separado:

- Desarrollo de una librería de procesado de contenido que permita adaptar los archivos de modelos 3D generados por los grafistas e importados usando el pipeline de contenido predeterminado.

- Implementación de los shaders necesarios para la obtención de la imagen final a partir de los modelos y las luces.

- Desarrollo de una capa de abstracción que permita configurar los distintos tipos de luces y cámara utilizados y utilizarlos con sencillez en una aplicación.

- Creación de un visor sencillo que sirva como demostración del uso de la tecnología desarrollada y poner ejemplos para distintos escenarios.

ETAPAS DEL DESARROLLO

Se pueden distinguir siete etapas en el desarrollo de este proyecto.

En la primera etapa se estudian las distintas tecnologías disponibles y el estado de cada una de ellas para determinar la viabilidad del proyecto. Del mismo modo, se realiza una investigación sobre las distintas técnicas de renderizado en diferido para elegir la más adecuada al proyecto.

La etapa segunda consiste en un análisis del problema, recogiendo los requisitos que el sistema deberá satisfacer. Al no haber un cliente, los requisitos se determinan desde el punto de vista de un desarrollador que fuera a usar nuestro sistema.

Una vez detallados los requisitos, en la tercera etapa se pasa a diseñar la arquitectura del sistema y los distintos componentes de ésta.

Tras todo el trabajo anterior ya es posible comenzar a implementar el sistema. En la cuarta etapa se implementa la librería de gestión de modelos, creando el procesado de estos y su carga en el sistema.

En una quinta etapa se implementa la arquitectura de render en cuestión, siendo capaces tras esto de mostrar un modelo renderizado en diferido con iluminación.

Para facilitar las pruebas y la demostración de los resultados, es necesaria la creación de un visor 3D sencillo en una sexta etapa.

Finalmente, este documento se elabora en una séptima etapa.

ESTRUCTURA DEL DOCUMENTO

Este apartado se destina a explicar brevemente los puntos desarrollados en los distintos capítulos del documento.

1. INTRODUCCIÓN

Pretende dar una visión global del proyecto, introduce los objetivos, las motivaciones y los términos usados en el documento para una mejor comprensión por parte del lector y explica brevemente el desarrollo.

2. ESTADO DE LA CUESTIÓN

Introduce el problema a abordar y los distintos tipos de arquitecturas de renderizado, explicando brevemente en que consiste cada una para familiarizar al lector con técnicas complejas que se usarán en el desarrollo.

También se analizan las características de algunos de los principales motores de desarrollo de videojuegos, tanto profesionales como independientes centrándose en XNA, el motor utilizado en el proyecto.

3. ANÁLISIS, DISEÑO E IMPLEMENTACIÓN

En este apartado se describe el proceso de desarrollo del proyecto. Primero se analiza el problema, obteniendo sus requisitos de usuario y de software. Después se explican brevemente las decisiones de diseño tomadas a la hora de elaborar la solución. Finalmente se detallan algunas partes esenciales de la implementación del sistema.

4. RESULTADOS OBTENIDOS

Se muestran imágenes obtenidas con el visor de distintas escenas. Se muestran por separado cada tipo de luz y algunas de sus propiedades, para observarlas más fácilmente.

5. TRABAJOS FUTUROS Y CONCLUSIONES

En el último apartado se habla, de forma subjetiva, de las conclusiones llegadas al término del proyecto, analizando sus ventajas y estudiando cómo podrían solucionarse los principales inconvenientes de este tipo de renderizado.

Además, propondremos una serie de técnicas que explotan las ventajas de esta arquitectura y podrían implementarse en el futuro.

ESTADO DE LA CUESTIÓN

CONCEPTOS BÁSICOS

Antes de entrar a evaluar los distintos tipos de arquitecturas de renderizado, es necesario presentar los distintos elementos necesarios para el renderizado, así como las etapas comunes a todas las arquitecturas.

ELEMENTOS DE UN OBJETO 3D

Aunque existen otras formas menos convencionales de representar objetos 3D por software, como por ejemplo con superficies formadas por curvas o subdividiendo el espacio en una fina rejilla de cubos (voxels), nos centraremos en la forma tradicional basada en polígonos que soportan las tarjetas aceleradoras de gráficos actuales.

Las mallas poligonales mostradas de esta forma están formadas por caras que unen vértices y que tienen distintos materiales.

- VÉRTICES:

Conjunto de puntos tridimensionales que conforman la forma del objeto. Como mínimo consta de un vector de posición, pero puede contener también información de la normal de la superficie, su tangente, coordenadas de mapeado de textura e incluso color.

- CARAS:

Polígonos (generalmente triángulos) que aproximan la superficie real del objeto a una forma más sencilla de representar.

Generalmente se almacenan como conjuntos de índices que apuntan a los vértices del objeto, que pueden ser compartidos por varias caras.

- MATERIALES:

El concepto de material hace referencia a todas las propiedades visuales de una cara. Dependiendo de la calidad del render puede almacenar desde algo tan sencillo como el color o la textura hasta algo complejo como un programa a ejecutar en la GPU (shader).

Algunas de las propiedades habituales son la textura para el color, la textura para la normal, la intensidad y el tipo del brillo (ya sea un parámetro o una textura) y la auto iluminación.

ETAPAS DE RENDERIZADO DE UN OBJETO 3D EN PANTALLA

Independientemente del software y el hardware utilizado existen una serie de etapas que es necesario realizar para pasar de una nube ordenada de vértices en 3D a su representación en pantalla.

En el hardware más moderno existen etapas adicionales que trabajan a nivel de geometría, creando nuevos vértices si es necesario, como pueden ser la teselación de malla o los geometry shaders, pero están fuera del objeto de este estudio.

- TRANSFORMACIÓN DE LOS VÉRTICES

El almacenamiento de los vértices que conforman la malla 3D suele ser relativo al centro del objeto. Por ello existen varias matrices por las que se debe transformar un vértice para calcular su posición en pantalla.

MATRIZ DE MUNDO:

Almacena la transformación del centro del objeto con respecto al mundo. Tener por separado esta información en lugar de guardar los vértices ya transformados en el mundo permite, por un lado, una fácil transformación del objeto con respecto al tiempo modificando solamente su matriz de mundo y, por otro lado, tener varias instancias del mismo objeto, cada una con su propia matriz de mundo.

MATRIZ DE VISTA:

Una vez el vértice está posicionado de forma definitiva en el mundo, es necesario calcular cuál es su posición con respecto al punto de vista que queremos utilizar, es decir, con respecto a la cámara que estamos usando. Esta matriz de transformación es común para todos los objetos que usen la misma cámara.

Al tratarse de transformaciones lineales es una práctica común el calcular una única vez por objeto la combinación de matriz mundo-vista para cada objeto y luego usar ésta para calcular cada vértice del objeto.

MATRIZ DE PROYECCIÓN:

Tras las etapas anteriores el vértice se encuentra en coordenadas 3D relativas a la cámara. Finalmente, es necesario calcular una posición en 2D para poder situarlo en la pantalla. La matriz utilizada para esto se llama matriz de proyección y puede ser ortográfica o con perspectiva, según cómo afecta la profundidad a la posición bidimensional.

- DESCARTE DEGEOMETRÍA

No todos los vértices transformados terminarán dentro del marco donde se representará la escena. La matriz de proyección establece 6 planos principales de *clipping*, uno a cada lado (*left* y *right*), otros dos arriba y abajo (*top* y *bottom*) y finalmente uno para los objetos detrás o demasiado cerca de la cámara (*near*) y otro para aquellos que están demasiado lejos (*far*).

Es posible también especificar planos adicionales para descartar forzadamente regiones que se saben ocultas (por ejemplo, detrás de un muro sólido).

Por lo general los cuerpos que se representan tienen volúmenes cerrados, por lo tanto no es necesario renderizar aquellas caras cuya normal no apunta hacia la cámara. El descarte de estas caras (*culling*) se produce también en esta etapa.

- RASTERIZADO DE LOS TRIÁNGULOS FORMADO POR LOS VÉRTICES

Cuando ya se conoce la posición de los distintos vértices que conforman una cara es posible determinar que pixeles de la pantalla se ven afectados por esta.

El rasterizador se encarga de generar la información adecuada para cada uno interpolando linealmente las propiedades de los vértices. Como mencionamos anteriormente, un vértice puede contener más información que la de posición. Toda esta información se interpolará linealmente del mismo modo.

- CÁLCULO DEL COLOR DEL PIXEL

Con toda la información de los vértices y las propiedades del material se genera la información de salida del pixel, consistente en un color por cada *rendertarget* asociado (generalmente solo uno).

La forma de calcular esta etapa es una de las principales diferencias entre los distintos métodos de render. Puede ser resultado de una formula general configurable o, más frecuentemente, calculado usando un programa específico (shader) que se ejecuta en el procesador gráfico. Para esto puede usar como entradas parámetros variables y texturas.

- DESCARTE DE PIXELES

Al igual que ocurre con los vértices, no todos los píxeles calculados acaban siendo mostrados en la pantalla. Aunque estos tienen siempre su posición fija dentro del marco, pueden ser descartados por otros motivos.

El principal motivo de descarte de un pixel es por profundidad. Existe un testeo configurable entre el valor de profundidad del pixel actual y del nuevo. Por lo general si el valor actual del pixel en el fondo tiene una profundidad menor, el nuevo valor se supone detrás y no se escribirá. Este comportamiento puede ser deshabilitado o modificado para conseguir otros efectos.

De forma similar, existe un mecanismo general que almacena un valor actual que puede ser testeado y usado para descartar un pixel. Se conoce como *stencil buffer* y almacena valores numéricos enteros, como por ejemplo el número de veces que ha sido escrito un determinado pixel en una pasada. Su uso es poco frecuente, pero es la base de técnicas como las sombras duras (*stencil shadows*).

Por último, la transparencia del color generado puede usarse para descartar el pixel. Si se ha configurado un umbral de transparencia y este no es superado, el pixel no se escribirá. Esta comprobación se conoce como *alpha test*.

- BLENDING DEL PIXEL CON EL FONDO

El color resultante del pixel se escribe en el fondo o *framebuffer*. Es posible combinar este pixel con el pixel actual del fondo usando distintas fórmulas que tienen en cuenta tanto el valor de color como de transparencia de ambos.

Además del relleno sólido (rellena el fondo con el valor íntegro del pixel que se ha calculado sin tener en cuenta lo que hubiera anteriormente), algunas de las combinaciones más usadas son el *alphablend* (interpola linealmente los valores de color de los pixeles en función de la transparencia del nuevo) y el *additive* (suma el valor calculado de color al valor actual del fondo).

Es importante mencionar esta etapa ya que, como veremos más adelante, una de las limitaciones del deferred rendering es que no puede trabajar con distintos tipos de blending.

ARQUITECTURAS DE RENDERIZADO

Actualmente no existe una forma única de presentar gráficos en pantalla. Así pues, coexisten en las aplicaciones actuales diversos métodos, cada uno con sus ventajas e inconvenientes, que comparten las etapas antes descritas pero las ejecutan de distintas formas.

PIPELINE FIJO

Es la arquitectura más antigua de las aquí expuestas. Es anterior a la existencia de programas ejecutables en el procesador gráfico pero en la actualidad coexiste con las arquitecturas más complejas como alternativa simple para aquellas etapas que no se requieran programar. Está empezando a desaparecer en algunas implementaciones y no es de extrañar que no sea utilizada en un futuro.

Las etapas se ejecutan de forma secuencial y de la misma forma para todos los vértices y pixeles procesados. Ofrece una serie de opciones (dependientes de la implementación) para personalizar el resultado pero es muy limitada en cuanto a la variedad en los resultados que se pueden conseguir.

PIPELINE PROGRAMABLE

Como respuesta a la necesidad de tratar algunos materiales de una forma distinta, surgió la arquitectura programable. En esta arquitectura algunas etapas pueden ser personalizadas con programas específicos escritos para ser ejecutados en el chip gráfico, llamados shaders.

Las posibilidades de estos shaders se han ido incrementando con el tiempo, a medida que mejoraba el hardware de los chips gráficos. Las distintas generaciones de shaders se agrupan bajo "shader models" que especifican los requisitos mínimos que debe soportar el hardware y las limitaciones que tendrá que asumir el programador. Además indican que etapas pueden ser programadas y cuáles no.

Existen muchas versiones distintas de shader models, cada una mejorando a la anterior con nuevas características. La versión que implementa el hardware más moderno en la actualidad es la versión 5.0.

Dentro de un shader model, según la etapa en la que se encuentran, los shaders pueden ser de varios tipos:

- Geometry shaders: Reciben como entrada un conjunto de vértices y aristas. Estos shaders pueden eliminar vértices inservibles o crear nuevos. Los vértices generados como salida de esta etapa son procesados individualmente según lo expuesto anteriormente.

- Vertex shaders: Tienen como entrada y salida un único vértice, pero no necesariamente con las mismas propiedades. Como mínimo, estos shaders deben realizar la transformación de coordenadas del vértice. Por lo tanto, la propiedad de posición es obligatoria tanto de entrada como de salida. Adicionalmente pueden incluirse otros cálculos en estos shaders y pasarlos a las siguientes etapas como propiedades del vértices de salida.

- Pixel shaders: La entrada de estos shaders son las propiedades recibidas de los vértices interpoladas linealmente por el rasterizador según la posición del pixel. Es responsabilidad de estos shaders calcular el color resultante del pixel. También permiten descartar el pixel si no cumple algunas condiciones. La salida de estos shaders es uno o más valores de color, uno por cada rendertarget de salida.

DEFERRED SHADING

Esta arquitectura se apoya en una arquitectura de pipeline programable ya que usa shaders. El término deferred shading hace referencia a una técnica de shading en espacio de pantalla. Se llama deferred (diferido) porque no se calcula el color del pixel en la primera pasada de render sino que se pospone a una segunda pasada.

En la primera pasada de render sólo se reúne la información requerida para el cálculo posterior del color del pixel. Posiciones, normales y materiales para cada superficie son almacenadas por separado en texturas llamadas geometry buffers o, más frecuentemente, g-buffers. Después de esto, se procesa la iluminación directa e indirecta de cada luz de la escena en cada pixel usando la información de los g-buffers y determinando el color final del pixel. Este cálculo se realiza ya en espacio de pantalla.

La principal ventaja del deferred shading es la separación de la geometría de la escena de la iluminación. Solo se requiere una única pasada de geometría y la iluminación de cada luz se calcula únicamente para los pixeles a los que afecta realmente. Esto permite renderizar muchas luces en una escena sin afectar seriamente al rendimiento. Otras ventajas de esta arquitectura son la simplificación del manejo de escenas complejas con muchas luces y la sencilla integración con otras técnicas de post-procesado.

Entre las desventajas del deferred shading encontramos la incapacidad de manejar la transparencia como parte del algoritmo, aunque puede manejarse por separado como se haría en otras arquitecturas. Otra importante desventaja es la forma unificada de calcular el color de los pixeles a partir de los valores de iluminación y la información contenida en los g-buffers. Si se quisieran utilizar shaders distintos con otras propiedades, estas deberían ser almacenadas en otros g-buffers, con el coste en memoria que esto implica.

DEFERRED LIGHTING

Finalmente, existe una variación de la arquitectura de deferred shading, llamada. Surge como alternativa para solucionar el problema de los distintos shaders en el deferred shading. En esta arquitectura se utilizan tres pasadas en lugar de dos.

La primera pasada es idéntica a la de la arquitectura anterior, pero utilizando solo los g-buffers necesarios para el cálculo de iluminación. En la segunda pasada, de una forma parecida a la anterior, se calcula la iluminación de cada pixel utilizando la información de los g-buffers, pero no se determina el color final del pixel. Se requiere por lo tanto una tercera pasada, en la que vuelve a procesarse la geometría por segunda vez, para con la información ya calculada de iluminación procesar los distintos materiales y shaders con todas sus propiedades originales.

La desventaja de este tipo de arquitectura, además de ser ligeramente más compleja, es la de necesitar dos pasadas de geometría en lugar de una sola.

MOTORES DE DESARROLLO DE JUEGOS INDEPENDIENTES

La oferta de motores de desarrollo de uso personal es hoy en día más variada y de calidad de lo que ha sido nunca. Las grandes empresas se han dado cuenta del potencial de este mercado y proporcionan alternativas de bajo coste con funcionalidad limitada.

Por lo general se trata de opciones más sencillas de utilizar pero aun así permiten conseguir resultados con aspecto profesional.

XNA

Una de las plataformas pioneras en ofrecer resultados de calidad con poco esfuerzo. Forma parte de una iniciativa de Microsoft para acercar el desarrollo de videojuegos al desarrollador no profesional.

Usa el lenguaje C# como lenguaje de programación, lo que le permite conseguir un alto rendimiento sin tener que preocuparse demasiado de detalles de bajo nivel como la gestión de la memoria. Se distribuye conjuntamente con el entorno de desarrollo gratuito Visual C# Express, muy similar a su hermano mayor Visual Studio.

Permite desarrollar juegos para PC, Xbox 360 y, en sus últimas versiones, también para Windows phone y Zune.

Su motor gráfico funciona sobre DirectX, pero no permite el acceso directo a todas sus características. Sacrifica flexibilidad a cambio de una gran sencillez de uso. No soporta de forma nativa el uso de deferred rendering,

La plataforma XNA fue anunciada por primera el 24 de marzo de 2004, en la GDC (Game Developers Conference) de San Jose, California. Desde entonces, han sido varias las versiones que se han publicado:

XNA GAME STUDIO EXPRESS

XNA Game Studio Express, la primera versión publicada de XNA Game Studio, estaba destinada para estudiantes, aficionados y desarrolladores independientes. Se distribuyó mediante descarga de forma gratuita.

Proporcionaba "starter kits" básicos para el desarrollo rápido de cierto tipo de géneros de videojuegos, como por ejemplo los de plataformas, estrategia en tiempo real y shooters en primera persona. Los desarrolladores podían crear juegos de Windows gratuitamente, pero para ejecutar el juego en la Xbox 360 tenían que pagar una suscripción anual de 99$ (o una de cuatro meses al precio de 49$) para ser admitidos en el Microsoft XNA Creator's Club.

La versión inicial no tenía forma de compartir binarios compilados con otros jugadores de Xbox 360, pero esto cambió en la edición "XNA Game Studio Express 1.0 Refresh" que hacia posible compilar binarios para Xbox 360 y compartirlos con otros miembros del Microsoft XNA Creator's Club.

La primera versión beta de XNA Game Studio Express fue publicada para descarga el 30 de agosto de 2006, seguida por una segunda versión el 1 de noviembre de 2006. Microsoft lanzó la versión final el día 11 de diciembre de 2006.

Meses más tarde, el 24 de abril de 2007, por el motivo mencionado anteriormente, se publicó la actualización XNA Game Studio Express 1.0 Refresh.

XNA GAME STUDIO 2.0

XNA Game Studio 2.0 fue publicada el 13 de diciembre de 2007. La principal característica que ofrecía era la posibilidad de ser utilizado con todas las versiones de Visual Studio 2005, incluida la versión gratuita Visual C# 2005 Express Edition.

También incorporaba una API de red usando Xbox Live tanto en PC como en Xbox 360 y un mejor manejo de dispositivos.

XNA GAME STUDIO 3.0

XNA Game Studio 3.0 (para Visual Studio 2008 o la versión gratuita Visual C# 2008 Express Edition) añadió la posibilidad de desarrollar videojuegos para la plataforma Zune (dispositivos MP4 de Microsoft). La beta fue publicada en septiembre de 2008 y la versión final el 30 de octubre del mismo año.

XNA Game Studio 3.0 incluía soporte para C# 3.0 y LINQ, junto con varias características nuevas como el modo Trial, que permitía a los desarrolladores añadir de forma sencilla una versión demostrativa del juego, soporte para las características multijugador de Xbox LIVE, como invitar amigos desde dentro del juego y la posibilidad de crear juegos que interactuaran entre las distintas plataformas soportadas (Windows, Xbox 360 y Zune).

XNA GAME STUDIO 3.1

XNA Game Studio 3.1 se publicó el 11 de junio de 2009.

Esta revisión incorporaba soporte para la reproducción de video, una API de audio actualizada y soporte para usar los avatares de Xbox 360 dentro de los juegos,

XNA GAME STUDIO 4.0

XNA Game Studio 4.0 se anunció y lanzó como "Community Technical Preview" en la GDC de marzo de 2010 y en su versión final el 16 de septiembre de 2010.

Esta versión añade una nueva plataforma, Windows phone, para aprovechar el mercado de la telefonía móvil. Además, proporciona efectos configurables, reproducción de audio con buffer, entrada multi-táctil, entrada de micrófono e integración con Visual Studio 2010.

Una actualización, llamada “XNA Game Studio 4.0 Refresh”, se lanzó el 6 de octubre de 2011. Esta actualización añadía soporte para la versión 7.5 de Windows Phone, soporte para Visual Basic y corregía algunos errores.

El 31 de enero de 2013 Microsoft anunció, mediante un email a sus MVP (Most Valuable Professionals) que se había dejado de desarrollar activamente XNA, por lo que no habrá nuevas versiones. El soporte para la versión actual cesará en abril de 2014.

UNITY

Unity es probablemente el motor de mayor popularidad en la actualidad. Su mayor virtud consiste en su fácil portabilidad, pudiendo desarrollar un proyecto para PC, tableta y móvil de una forma casi transparente. Además de esto cabe destacar la sencillez de uso de su sistema basado en componentes y la fuerte comunidad de usuarios que tiene detrás.

Integra, además de un motor gráfico, todo lo necesario para el desarrollo de un juego. Incluye el motor de física nVidia PhysX para colisiones, sistema de audio basado en FMOD y muchas otras funcionalidades.

Permite incluir scripts en su propio lenguaje (Unity Script), en C# o en Boo (lenguaje con sintaxis parecida a Python).

La característica principal de su motor gráfico es igualmente su portabilidad. Funciona bajo Direct3D, OpenGL, OpenGLES e incluso existen implementaciones en soluciones propietarias.

El aspecto negativo de esta característica es la generalidad de su motor gráfico. Aunque si bien es muy completo (especialmente en su versión de pago, Unity Pro) no aprovecha al máximo las posibilidades de cada plataforma. Del mismo modo, aunque permite expandir sus características mediante nuevos shaders, es necesario pensar en las distintas plataformas a la hora de hacerlo.

En la práctica los juegos hechos con Unity se centran más en explorar los aspectos jugables que en ofrecer un aspecto visual sorprendente, por lo que las posibilidades del motor son más que suficientes.

UDK

UDK (Unreal Development Kit) es la alternativa gratuita del popular motor de juegos Unreal Engine, propiedad de Epic Games. Está disponible desde noviembre de 2009.

Esta versión algo reducida del motor permite incluso la comercialización de los juegos con unas condiciones muy asequibles ($99 iniciales y 25% de royalties a partir de los $50.000 de beneficios).

Mantiene las ventajas de portabilidad de la versión comercial pero también su complejidad. Es una buena alternativa para aquellos que buscan un resultado muy similar al de un producto comercial (a costa de una mayor inversión en tiempo de desarrollo). Además, al ser parecido a la versión de pago, es una experiencia a valorar a la hora de entrar a formar parte de un equipo que trabaje con Unreal Engine.

3D GAME STUDIO

3D Game Studio o 3DGS es un sistema complete de desarrollo de videojuegos que permite al usuario crear juegos o aplicaciones 3D y publicarlas sin royalties.

Se compone de un editor de modelos y terrenos, un editor de niveles, un editor y depurador de scripts y un motor gráfico propio.

Para hacerlo más asequible a usuarios principiantes incluye también una colección de modelos, texturas y plantillas de videojuegos para crear shooters o juegos de rol sin necesidad de programar.

Existen tres tipos de licencias, en función del tipo de usuario al que va dirigido. La más sencilla (licencia *beginner*) se vale únicamente de los editores y las plantillas para hacer juegos sencillos, pero tiene otras dos que permiten programar cualquier tipo de juego, una usando su propio lenguaje de script integrado llamado Lite-C (licencia *advanced*) y otra que puede integrarse con un entorno de desarrollo externo como Visual Studio o Borland Delphi (licencia *professional*).

Su punto fuerte es su sencillez, lo que permite crear una comunidad de usuarios que, a medida que aprenden, siguen utilizando el producto en sus versiones más avanzadas.

MOTORES DE DESARROLLO DE JUEGOS PROFESIONALES

CRYENGINE

CryEngine fue concebido como una demo tecnológica creada por Crytek para nVidia. Tras ver su potencial, decidieron transformarlo en un videojuego, utilizándolo por primera vez en el shooter en primera persona *FarCry* en 2004.

A día de hoy, tras varias versiones, se trata probablemente del motor de videojuegos con más prestigio en cuanto a tecnología. Tras tres versiones numeradas, recientemente se anunció que la próxima versión sería llamada simplemente CryEngine. Esta versión del motor soporta ya la nueva generación de consolas.

Además de una colección de features y técnicas novedosas demasiado grande como para listarla aquí, presume principalmente de una capacidad de edición directamente sobre la plataforma final que pocos motores son capaces de conseguir.

Aunque los juegos que han sido desarrollados con el no han sido aclamados por la crítica, algunos como el Crysis han sido referente gráfico durante muchos años.

Algunos de los juegos más conocidos desarrollados con este motor son: la saga FarCry, la saga Crysis y, recientemente, Ryse: Son of Rome.

UNREAL ENGINE

Unreal Engine es el motor de videojuegos de Epic Games. Fue utilizado por primera vez en el Unreal en 1998. Aunque inicialmente fue desarrollado para shooters en primera persona en la actualidad se utiliza ampliamente y para cualquier género de videojuego.

Una de sus virtudes más conocidas es la de disponer de un editor visual de scriptado llamado Kismet, con el cual los diseñadores pueden crear situaciones sin necesidad de un programador.

Gracias a su gran portabilidad (funciona en prácticamente todas las plataformas comerciales actuales) y a sus muchas herramientas de desarrollo es el motor más utilizado por terceros.

La versión actual más extendida es la 3, pero pronto será sustituida por la cuarta versión.

Existen infinidad de videojuegos de todo tipo desarrollados con este motor, pero los más conocidos (además de la propia saga Unreal) son los de la saga Gears of War, desarrollada por la propia. Como ejemplos de juegos desarrollados con este motor por terceros merece la pena mencionar la saga Bioshock y la saga MassEffect.

RAGE

Aunque es muy poco conocido por su nombre (de hecho la mayoría lo asocia erróneamente al videojuego Rage, producido por Id Software), es uno de los motores con mejores resultados en la actualidad.

Desarrollado por la multinacional Rockstar (especialmente por su estudio de San Diego), es un motor menos extendido ya que no se licencia a terceros, pero se utiliza en todos los juegos de la compañía.

Es un motor relativamente nuevo, aunque haya evolucionado de motores anteriores de la compañía. El primer juego desarrollado con él fue "Rockstar presents Table Tennis", un juego de ping pong, pero fue con el Grand Theft Auto 4 con el que ganó popularidad. Incorpora middleware de terceros como el popular Euphoria para el sistema de animaciones.

Aunque son pocos los juegos desarrollados hasta el momento con este motor, cuenta con grandes éxitos como las dos últimas entregas de la saga Grand Theft Auto (IV y V) o el aclamado juego del oeste Red Dead Redemption.

FROSTBITE

Al igual que el anterior, se trata de un motor propio, en este caso de Electronic Arts. Fue desarrollado en EA Digital Illusions (DICE) principalmente para su saga Battlefield.

Destaca principalmente su tecnología de destrucción de objetos y su sistema de audio HDR.

Además de la saga Battlefield, otros juegos de la compañía utilizan este motor, como por ejemplo el próximo Mirror's Edge.

OTROS

Aunque solamente hemos citado cuatro ejemplos muy conocidos, la variedad de motores de videojuegos utilizados en los juegos de última generación es muy amplia.

No todos los motores son propiedad exclusiva de grandes multinacionales, algunos estudios más pequeños tienen sus propios motores que no tienen nada que envidiar a los más conocidos, como por ejemplo el utilizado por los polacos CDProjektRed en su saga Witcher.

En España cabe destacar la tecnología utilizada por las empresas Mercury Steam Entertainment en consolas y PC o Digital Legends en plataformas móviles, que compiten entre los mejores.

ANÁLISIS, DISEÑO E IMPLEMENTACIÓN

ANÁLISIS

REQUISITOS DE USUARIO

Este apartado recoge los requisitos de usuario, que se dividirán en dos categorías: requisitos de usuario de capacidad y requisitos de usuario de restricción.

Los requisitos serán recogidos en tablas cuyo formato incluirá los siguientes campos:

- Nombre: Nombre identificativo del requisito. Ha de ser unívoco y se utilizará a modo de resumen de dicho requisito. Deberá ser lo suficientemente descriptivo.

- ID: Identificador unívoco del requisito de usuario. Su nomenclatura seguirá el formato: UR_X_NN, donde:

*X se sustituirá por 'C' para los requisitos de usuario de capacidad, o 'R' para los requisitos de restricción.

*NN será sustituido por el número de requisito dentro de su categoría, comenzando en 01 e incrementándose a cada nuevo requisito.

- Descripción: Descripción detallada del requisito. Todos los detalles han de incluirse aquí para no dar lugar a ambigüedades.

- Prioridad: Indica la prioridad de implementación del requisito. Facilita la organización y la distribución de carga de trabajo durante el desarrollo del proyecto. Puede tomar los valores *Baja*, *Media* y *Alta*.

- Necesidad: Indica como de necesaria es la introducción del requisito en el proyecto. Puede tomar los valores *Esencial* y *Deseable.*

- **Estabilidad:** Indica si el requisito puede ser objeto de modificaciones durante el ciclo de vida del proyecto, o si por el contrario se trata de un requisito que no ha de sufrir variaciones. El campo puede tomar los valores *Estable* e *Inestable,* dependiendo de si se da el segundo o primer caso, respectivamente.

- **Verificabilidad:** Establece con qué facilidad se puede comprobar que el requisito haya sido introducido en el proyecto. Los valores que puede tomar son *Alta*, *Media* y *Baja*.

REQUISITOS DE CAPACIDAD

Los requisitos de capacidad indican qué funciones puede o tiene que hacer la aplicación para cumplir los propósitos requeridos para el sistema.

NOMBRE	Renderizado de modelos simples		ID	UR_C_01
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	El visor debe ser capaz de cargar y mostrar modelos 3D sencillos, con un único material y un poligonaje bajo.			

Tabla 1. Requisito de usuario UR_C_01

NOMBRE	Renderizado de modelos complejos		ID	UR_C_02
PRIORIDAD	Media	NECESIDAD	Deseable	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	El visor debería ser capaz de cargar y mostrar modelos 3D complejos, con múltiples materiales y con un número de polígonos aceptable, según la potencia del procesador gráfico de la tarjeta.			

Tabla 2. Requisito de usuario UR_C_02

NOMBRE	Uso de luz de ambiente		ID	UR_C_03
PRIORIDAD	Media	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	La iluminación de todos los objetos debe verse afectada por igual por un color de base cuyo propósito es el de simular la luz dispersa reflejada por las superficies de la escena. El color de esta luz debe ser modificable dinámicamente y puede ser de color negro (no aportar iluminación).			

Tabla 3. Requisito de usuario UR_C_03

NOMBRE	Uso de luces omnidireccionales		ID	UR_C_04
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	La escena deberá utilizar como mínimo puntos de luz omnidireccionales como fuentes de luz. La luz deberá atenuarse con la distancia, siendo posible configurar la distancia a partir de la cual la luz empieza a atenuarse y la distancia máxima a partir de la cual la fuente de luz no aportará iluminación alguna.			

Tabla 4. Requisito de usuario UR_C_04

NOMBRE	Uso de luces de foco		ID	UR_C_05
PRIORIDAD	Media	NECESIDAD	Deseable	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	La escena debe poder incluir luces de foco. Las luces de foco deben tener una distancia de atenuación configurable, con un inicio y final de atenuación. De forma similar, también debe ser posible atenuar el efecto según el ángulo relativo a la dirección del foco, con un ángulo de inicio de atenuación y un ángulo máximo del foco que nunca debe superar los 180°.			

Tabla 5. Requisito de usuario UR_C_05

NOMBRE	Uso de luces direccionales		ID	UR_C_06
PRIORIDAD	Baja	NECESIDAD	Deseable	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	La escena debería poder usar luces direccionales. Este tipo de luces afectarán a toda la escena independientemente de la distancia a la que se encuentren, ya que simulan la luz emitida por fuentes de luz situadas lo suficientemente lejos como para considerarlas en el infinito.			

Tabla 6. Requisito de usuario UR_C_06

NOMBRE	Iluminación con múltiples luces		ID	UR_C_07
PRIORIDAD	Media	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	Ya que la principal ventaja de esta arquitectura de renderizado es la posibilidad de utilizar un alto número de luces, el visor permitirá cargar y renderizar múltiples luces en una sola escena, pudiendo afectar todas ellas a todos los objetos.			

Tabla 7. Requisito de usuario UR_C_07

NOMBRE	Luces coloreadas		ID	UR_C_08
PRIORIDAD	Baja	NECESIDAD	Deseable	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	Todos los tipos de luces podrán aportar iluminación con color.			

Tabla 8. Requisito de usuario UR_C_08

NOMBRE	Luces dinámicas		ID	UR_C_09
PRIORIDAD	Baja	NECESIDAD	Deseable	
ESTABILIDAD	Estable	VERIFICABILIDAD	Media	
DESCRIPCIÓN	Las propiedades de las luces pueden cambiar en tiempo real. Los parámetros configurables (dependiendo del tipo de luz) serán: - Color - Posición (direccional, foco y omnidireccional) - Dirección (direccional y foco) - Atenuación por distancia (foco y omnidireccional) - Atenuación angular (foco)			

Tabla 9. Requisito de usuario UR_C_09

NOMBRE	Cambio de parámetros de cámara		ID	UR_C_10
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	Los parámetros de la cámara podrán cambiarse para enfocar la escena desde distintas perspectivas. Los parámetros modificables serán: - Posición - Ángulo - FOV			

Tabla 10. Requisito de usuario UR_C_10

NOMBRE	Texturas de color		ID	UR_C_11
PRIORIDAD	Media	NECESIDAD	Deseable	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	Los objetos obtendrán su color de una textura en lugar de usar un color fijo.			

Tabla 11. Requisito de usuario UR_C_11

NOMBRE	Texturas de normal		ID	UR_C_12
PRIORIDAD	Baja	NECESIDAD	Deseable	
ESTABILIDAD	Estable	VERIFICABILIDAD	Media	
DESCRIPCIÓN	La superficie de los objetos podrá simular relieve utilizando texturas de mapa de normales. Si el objeto no especifica una textura de mapa de normales se considerará solamente la normal de la superficie.			

Tabla 12. Requisito de usuario UR_C_12

NOMBRE	Texturas de reflexión especular		ID	UR_C_13
PRIORIDAD	Baja	NECESIDAD	Deseable	
ESTABILIDAD	Estable	VERIFICABILIDAD	Media	
DESCRIPCIÓN	El brillo o reflexión especular de los objetos se determinará utilizando texturas. Los canales RGB almacenarán el color de la reflexión según el material y el canal Alpha almacenará el coeficiente para el cálculo del tipo de reflexión.			

Tabla 13. Requisito de usuario UR_C_13

REQUISITOS DE RESTRICCIÓN

Los requisitos de restricción indican qué limitaciones tiene la aplicación. Al ser un visor de funcionalidad muy limitada, existen muy pocos requisitos de este tipo.

NOMBRE	Imposibilidad de interrupción del renderizado		ID	UR_R_01
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	El visor deberá renderizar los modelos en todas las circunstancias, no debiendo interrumpir su función sin la intervención del usuario.			

Tabla 14. Requisito de usuario UR_R_01

NOMBRE	Imposibilidad de renderizar luces con valor nulo		ID	UR_R_02
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	El visor no deberá incluir en la fase de dibujado de luces aquellas fuentes de luz que posean color negro o intensidad cero.			

Tabla 15. Requisito de usuario UR_R_02

REQUISITOS SOFTWARE

Este apartado recoge los requisitos software del proyecto, que de nuevo se dividirán en dos categorías: funcionales y no funcionales.

De forma similar a lo que ocurría con los requisitos de usuario, los requisitos software serán recogidos en tablas con los mismos campos con las siguientes diferencias:

- ID: Identificador unívoco del requisito de usuario. Su nomenclatura seguirá el formato: SR_X_NN, donde:

*X será sustituido por 'F' en caso de tratarse de un requisito funcional, o 'N' si no lo fuera.

*NN será sustituido por el número de requisito dentro de su categoría, comenzando en 01 e incrementándose a cada nuevo requisito.

- Dependencias: Indica qué requisito de usuario se evalúa o implementa con dicho requisito software. Cada requisito de usuario tiene que estar cubierto por uno o varios requisitos software.

NOMBRE	**Cargar modelo**		ID	**SR_F_01**
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	El listado de contenido permitirá indicar que un modelo se renderizará utilizando nuestro sistema de deferred shading. De forma paralela, se facilitará una función de carga de modelos de este tipo.			
DEPENDENCIAS	**UR_C_01, UR_C_02**			

Tabla 16. Requisito software SR_F_01

NOMBRE	**Configurar luz de ambiente**		ID	**SR_F_02**
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	La interfaz facilitará una propiedad para establecer el color de la luz de ambiente.			
DEPENDENCIAS	**UR_C_03**			

Tabla 17. Requisito software SR_F_02

NOMBRE	Añadir luz omnidireccional		ID	SR_F_03
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	Existirá una clase para representar luces de tipo omnidireccional, con las siguientes propiedades: - Posición - Color - Inicio de atenuación - Fin de atenuación			
DEPENDENCIAS	**UR_C_04, UR_C_07, UR_C_08, UR_C_09**			

Tabla 18. Requisito software SR_F_03

NOMBRE	Añadir luz direccional		ID	SR_F_04
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	Existirá una clase para representar luces de tipo direccional, con las siguientes propiedades: - Posición - Dirección - Color			
DEPENDENCIAS	**UR_C_05, UR_C_07, UR_C_08, UR_C_09**			

Tabla 19. Requisito software SR_F_04

NOMBRE	Añadir luz de foco		ID	SR_F_05
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	Existirá una clase para representar luces de tipo omnidireccional, con las siguientes propiedades: - Posición - Dirección - Color - Inicio de atenuación - Fin de atenuación - Inicio de atenuación angular - Fin de atenuación angular			
DEPENDENCIAS	**UR_C_06, UR_C_07, UR_C_08, UR_C_09**			

Tabla 20. Requisito software SR_F_05

NOMBRE	Configurar cámara		ID	SR_F_06
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	La interfaz permitirá acceder y modificar los valores de la cámara. En concreto, las propiedades mínimas serán: - Posición - Rotación - FOV			
DEPENDENCIAS	**UR_C_10**			

Tabla 21. Requisito software SR_F_06

NOMBRE	Carga de textura de color		ID	SR_F_07
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	El preprocesador de modelos comprobará la existencia de textura de color y la cargará, haciéndola accesible mediante la propiedad correspondiente.			
DEPENDENCIAS	**UR_C_11**			

Tabla 22. Requisito software SR_F_07

NOMBRE	Carga de textura de normal		ID	SR_F_08
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	El preprocesador de modelos comprobará la existencia de textura de normal y la cargará, haciéndola accesible mediante la propiedad correspondiente. En caso de no existir, se cargará una textura por defecto cuyo valor es equivalente a una superficie plana.			
DEPENDENCIAS	**UR_C_12**			

Tabla 23. Requisito software SR_F_08

NOMBRE	Carga de textura de reflexión especular		ID	SR_F_09
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	El preprocesador de modelos comprobará la existencia de textura de reflexión especular y la cargará, haciéndola accesible mediante la propiedad correspondiente. En caso de no existir, se cargará una textura por defecto cuyo valor es equivalente a una superficie de brillo blanco poco brillante.			
DEPENDENCIAS	**UR_C_13**			

Tabla 24. Requisito software SR_F_09

NOMBRE	Rendimiento adecuado		ID	SR_N_01
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	El rendimiento obtenido usando este tipo de renderizado no debe ser muy inferior al que se habría conseguido en una escena similar utilizando el renderizado por defecto de XNA.			
DEPENDENCIAS	**UR_R_01**			

Tabla 25. Requisito software SR_N_01

NOMBRE	Soporte de múltiples rendertargets		ID	SR_N_02
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	Para poder ejecutar adecuadamente el renderizado en diferido, será necesaria una tarjeta gráfica con soporte de múltiples rendertargets (MRT) y texturas de tipo flotante. La mayoría de las tarjetas actuales lo soportan pero el mínimo requerido es, para las tarjetas ATI un modelo Radeon 9500 o superior y para las nVidia una tarjeta de la serie 6000 o superior.			
DEPENDENCIAS	**UR_R_01**			

Tabla 26. Requisito software SR_N_02

NOMBRE	Descarte de luces con valor nulo		ID	SR_N_03
PRIORIDAD	Alta	NECESIDAD	Esencial	
ESTABILIDAD	Estable	VERIFICABILIDAD	Alta	
DESCRIPCIÓN	En la fase de pintado de luces se omitirán las luces cuya intensidad sea cero o su color negro.			
DEPENDENCIAS	**UR_R_02**			

Tabla 27. Requisito software SR_N_03

MATRIZ DE TRAZABILIDAD

Una vez recogidos y analizados tanto los requisitos de usuario como los requisitos software, ha de comprobarse que todo requisito de usuario esté contemplado por al menos un requisito software.

Para hacer esta comprobación, se utilizan matrices de trazabilidad. Una matriz de trazabilidad tiene en su eje vertical el conjunto de identificadores de requisitos de usuario, tanto de capacidad como de restricción, y en su eje horizontal, los identificadores de requisitos software.

Los requisitos software tienen un campo llamado *Dependencias* que indica el requisito de usuario evaluado o implementado por el requisito software en cuestión. Siendo así, la matriz de trazabilidad marcará cada requisito de usuario contemplado por un requisito software concreto, siempre y cuando este haya sido asignado en el campo *Dependencias* correspondiente.

	SR_F_01	SR_F_02	SR_F_03	SR_F_04	SR_F_05	SR_F_06	SR_F_07	SR_F_08	SR_F_09	SR_N_01	SR_N_02	SR_N_03
UR_C_01	X											
UR_C_02	X											
UR_C_03		X										
UR_C_04			X									
UR_C_05				X								
UR_C_06					X							
UR_C_07			X	X	X							
UR_C_08			X	X	X							
UR_C_09			X	X	X							
UR_C_10						X						
UR_C_11							X					
UR_C_12								X				
UR_C_13									X			
UR_R_01										X	X	
UR_R_02												X

Tabla 28. Matriz de trazabilidad

DISEÑO

ARQUITECTURA DE XNA

Antes de empezar a diseñar nuestro sistema es necesario hacer un estudio de la arquitectura del motor que vamos a utilizar.

La estructura de XNA se divide en varias capas. Las capas más bajas implementan la funcionalidad para que sea utilizable en las distintas plataformas. En la capa más alta se encuentran los starter kits para los distintos tipos de juegos tradicionales y los componentes creados por la comunidad. Es en esta capa donde se situará nuestro sistema.

Ilustración1.Framework de XNA

El asistente de creación de proyectos generará por nosotros una estructura genérica que comparten todos los proyectos de XNA. Esta estructura crea entre otras una clase que hereda de la clase Game, donde nosotros empezaremos a implementar nuestro sistema.

El ciclo de vida de la clase Game consta de varios pasos:

- Nada más ser ejecutado, pasa por una fase de inicialización.

- Después, se pasa por una fase de carga de contenido, donde el visor deberá cargar los datos de la escena.

- Cuando todos los recursos están ya disponibles, se entra en el bucle principal del juego. Este bucle comprende dos pasos del ciclo de vida. En este paso, se actualizan todos los componentes del juego contenidos en Game.

- Una vez actualizados, los componentes que sean dibujables, se dibujarán en pantalla. Tras mostrar el resultado, y mientras siga la ejecución normal, se vuelve al paso anterior. Si el usuario ha indicado que quiere acabar el juego, pasa al último paso.

- En el último paso se descarga todo el contenido y se liberan todos los recursos utilizados.

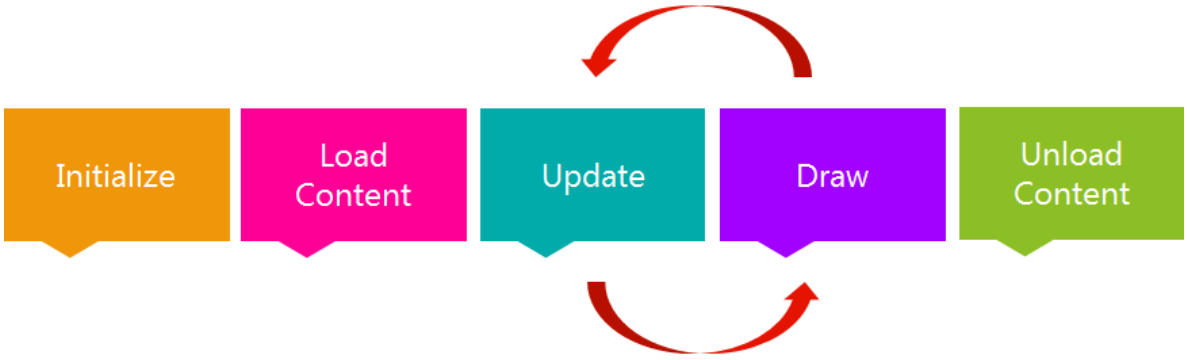


Ilustración 2. Estructura del loop principal de la clase Game

Para incorporar a este ciclo de vida nuestro sistema de renderizado lo más adecuado es hacerlo en forma de componente dibujable (DrawableComponent). De este modo, el juego se encargará de llamar a los métodos de actualización y pintado cuando corresponda, integrándolo perfectamente con otros componentes que pudiera necesitar el juego.

Nuestro componente almacenará la escena como una colección de modelos, una colección de luces y una cámara.

En el paso de carga de recursos, añadiremos a nuestra escena las luces y modelos que creamos oportuno, después la actualización realizará los cálculos que sean necesarios y finalmente la escena se pintará en la fase de dibujado.

A continuación veremos cómo deberemos manejar estos recursos para poder cargarlos en nuestra aplicación.

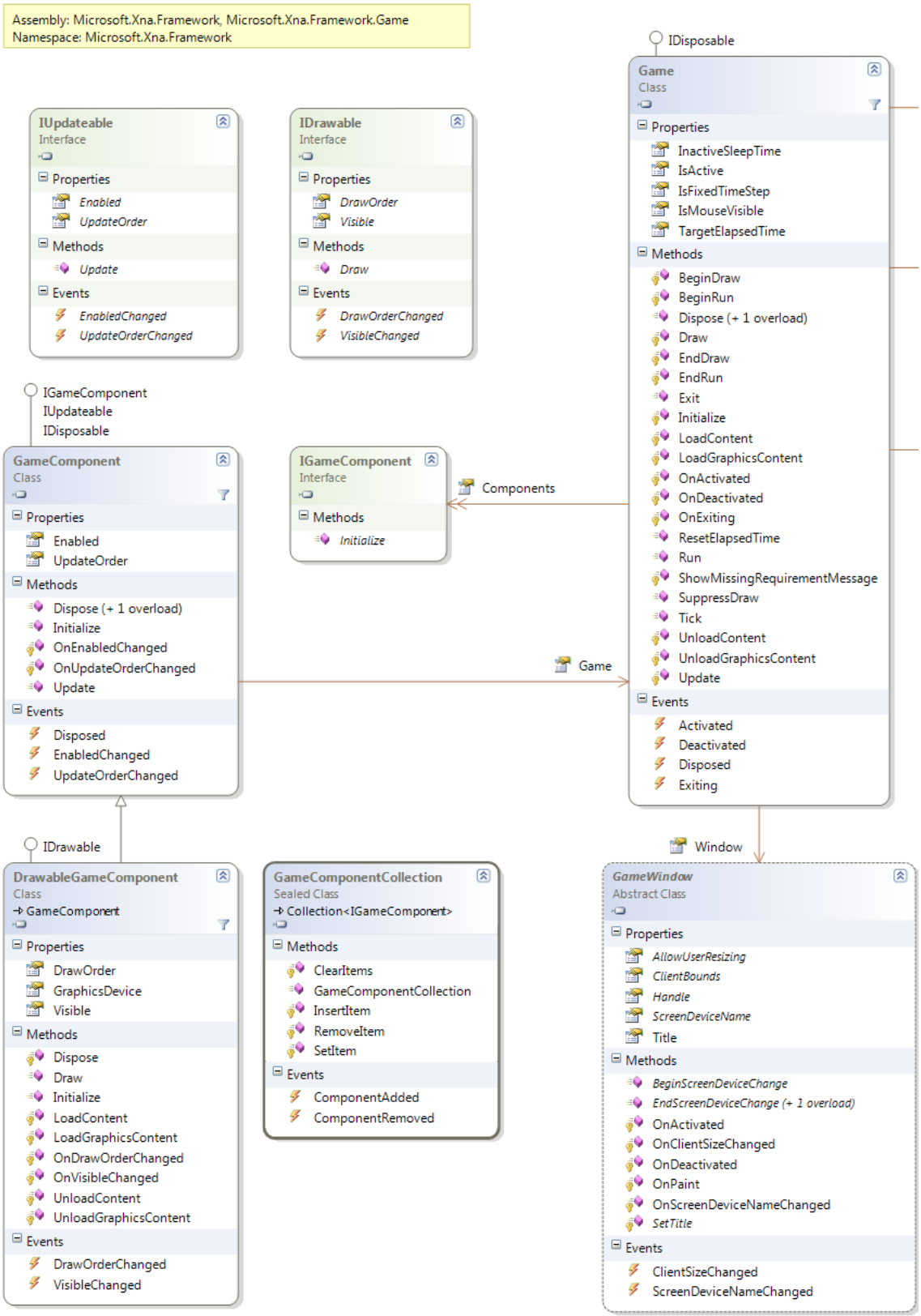


Ilustración 3. Diagrama de clases del sistema de componentes

PROCESADO DEL CONTENIDO

Antes de poder dibujar los modelos con nuestro shader personalizado, es necesario procesarlos para sustituir su shader por defecto por el nuestro.

Para ello, trabajaremos en un proyecto aparte no incluido en el visor de render que se enlaza al pipeline de contenido de XNA. El proyecto será del tipo “Content Pipeline Extension Library” existente en los proyectos predefinidos de XNA.

Existen distintos tipos de clases que se pueden extender dentro del content pipeline, como por ejemplo para importar nuevos formatos o tipos de assets. En nuestro caso queremos utilizar los importadores de modelos normales pero queremos procesarlos antes de que sea almacenado el binario que se utilizará durante el juego.

Para estos casos existe el template abstracto *ContentProcessor<TInput, TOutput>* que indica que tipo tomará como entrada y cuál será el resultado tras ser procesado. Esta clase junto con su *ContentProcessorContext* proporcionan todos los mecanismos necesarios para tratar el archivo fuente y añadir nuevas dependencias si fuera necesario.

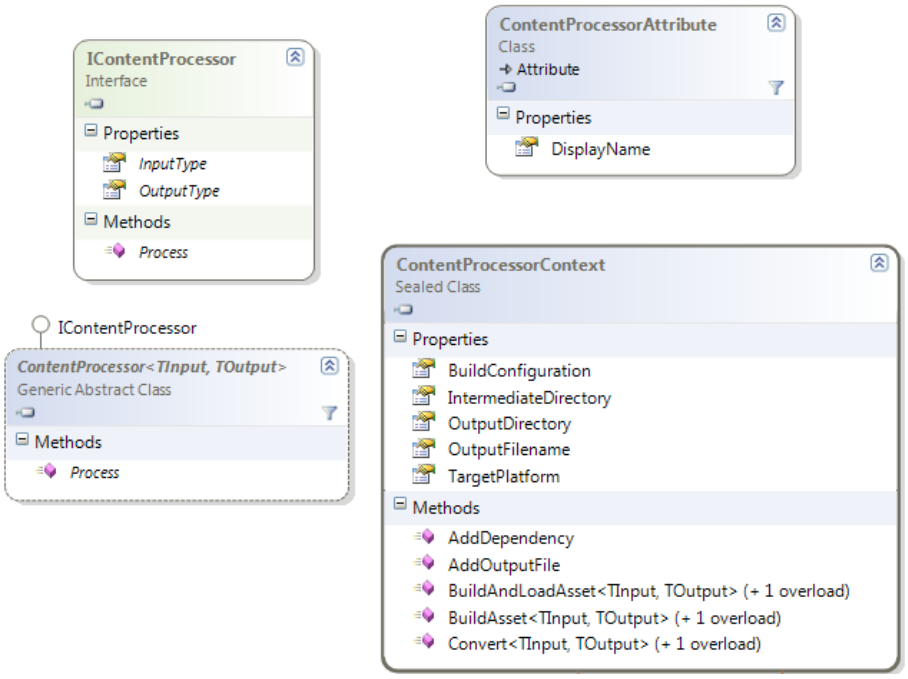


Ilustración 4. Diagrama de clases de ContentProcessor y su contexto

Por suerte, existen ya procesadores de contenido predefinidos para todos los tipos de contenido básicos. En la siguiente figura pueden verse todos ellos.

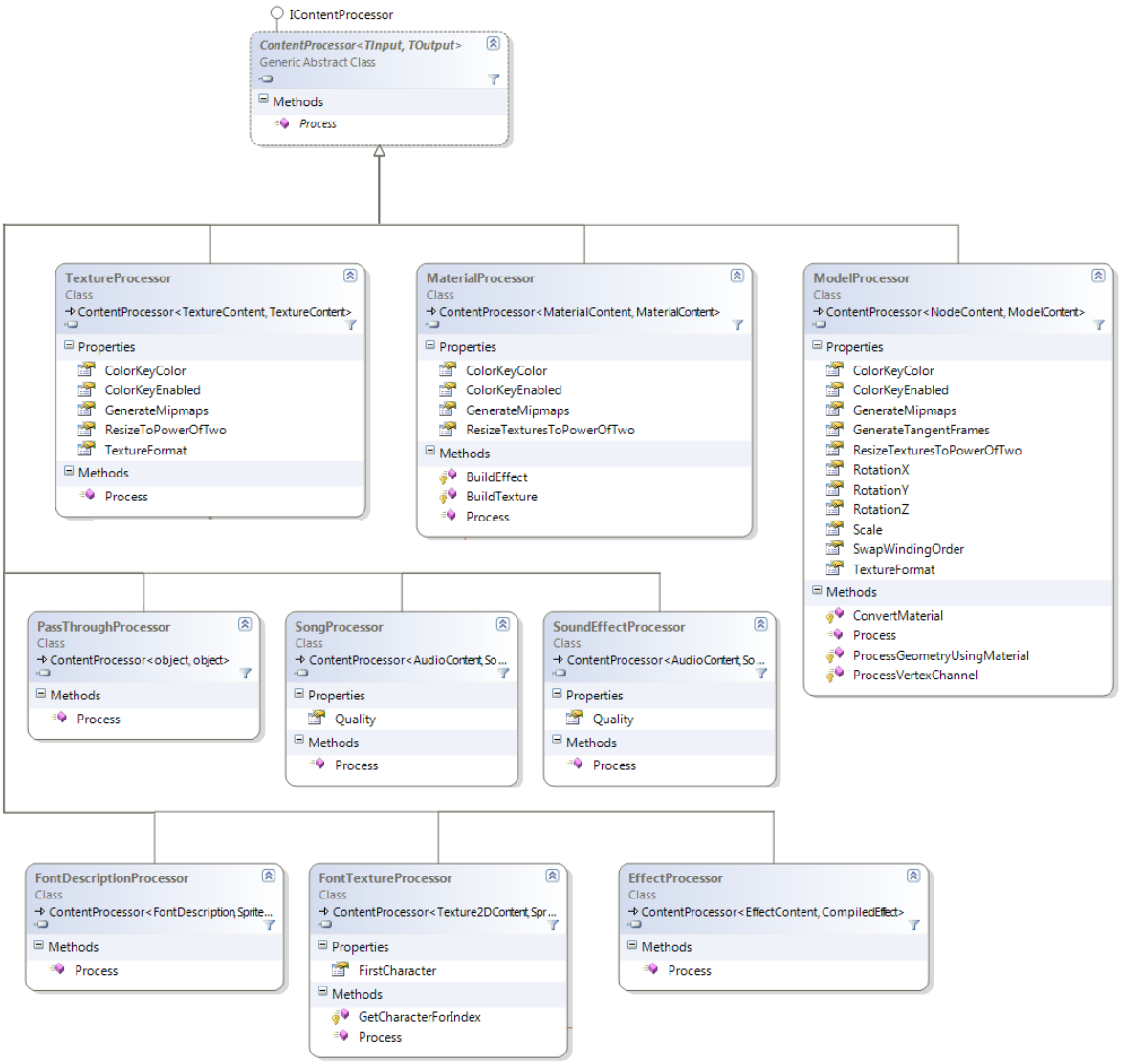


Ilustración 5. Diagrama de clases de los ContentProcessors predefinidos

Nosotros nos centraremos en los procesadores de contenido de modelos y materiales. Ha sido necesario crear uno de cada tipo para el procesamiento adecuado de los modelos.

La forma de indicar que el modelo usará nuestro procesador de contenido será seleccionarlo de la lista que aparece en las propiedades del modelo, dentro del explorador de la solución.

PROCESADOR DE MODELOS

Por un lado, el procesador de modelos redirige el procesador de materiales que se va a usar, sustituyendo el procesador por defecto por el nuestro. Para hacer esto ha sido suficiente sobrescribir el método que convierte los materiales para que, en lugar de utilizar la implementación de la clase base, convierta los materiales usando explícitamente nuestro procesador.

Además de esto, como en nuestro visor queremos usar normal mapping para dar más nivel de detalle al volumen mediante una textura de normales, ha sido necesario sobrescribir la propiedad *GenerateTangentFrames* para que sea cierta para todos los modelos que usen nuestro procesador de modelos.

Por último, aunque no está relacionado directamente con el motor de render, se ha aprovechado para procesar la información de los huesos para utilizar animación por *skinning*.

PROCESADOR DE MATERIALES

De forma paralela al procesador de modelos, el procesador de materiales ira convirtiendo uno a uno cada material contenido en los modelos que usen nuestro pipeline de contenido personalizado.

El primer paso, como habíamos mencionado anteriormente, es cambiar el shader por defecto por el nuestro especializado. La forma de hacerlo es tan sencilla como cambiar la propiedad Effect para hacer referencia al nuevo shader que hemos creado. El contenido de este shader será explicado con detalle más adelante.

Al igual que ocurre con el procesador de modelos, hemos aprovechado la creación de esta etapa de procesado de los materiales para realizar aquí algunos pasos que pueden ser realizados en tiempo de compilación.

Como veremos cuando analicemos el contenido del nuevo shader, hemos creído conveniente posibilitar el uso de texturas tanto para la información de normales (*normal mapping*) como para almacenar la información de brillo especular. En este paso se comprueba si existe una textura con el sufijo “-n” para la información de normales y “-s” para la información de brillo especular y, si existen, se cargan como dependencias. Si no existe alguna de estas texturas, se reemplaza por una textura vacía (el formato interno de estas texturas es distinto según el tipo) para que el shader no tenga que preocuparse de comprobar la existencia.

CONTENIDO ADICIONAL

Aunque no forma parte de la arquitectura de renderizado, es necesario dar soporte a la gestión de otras clases que conformarán las escenas que se probarán en el visor. Dichas escenas se especificarán en documentos XML que contendrán los datos de la cámara, las luces y los modelos contenidos.

Una vez más, el pipeline de contenido de XNA nos simplifica mucho la labor de cargar este contenido. Uno de los importadores que proporciona el pipeline permite leer cualquier clase desde un XML. Este importador detecta la clase buscando el atributo “Type” y lee las propiedades como etiquetas anidadas dentro del objeto.

Por lo tanto, si nos servimos de este mecanismo, bastará con implementar para cada tipo de contenido adicional una clase de tipo Content que especifique los datos que se almacenarán, una clase de tipo Writer que defina la forma en la que estos datos se almacenan y una clase de tipo Reader que lea los datos siguiendo el mismo formato. Para las clases de tipo Writer y Reader existen unas clases base que ofrecen la funcionalidad necesaria para escribir al binario y leer de él todos los tipos de dato básicos.

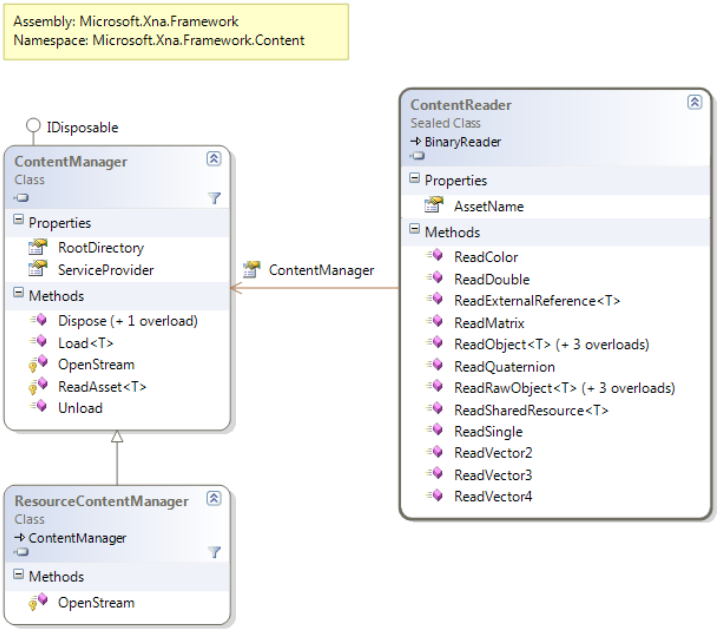


Ilustración 6. Diagrama de clases para la lectura de contenido

Las clases de tipo Content y las de tipo Writer pertenecerán a la librería de contenido, mientras que las de tipo Reader deben pertenecer a la librería principal ya que su código es el único que no se ejecutará en tiempo de compilación.

La clase que define el contenido de la cámara tendrá las siguientes propiedades:

- Posición: Vector de 3 componentes que define la posición en el mundo de la cámara.

- Objetivo: Vector de 3 componentes que define la posición a la que se orienta.

- FOV: Valor en radianes que define el ángulo de visión de la cámara en su eje horizontal.

- Near: Distancia del plano de descarte por cercanía.

- Far: Distancia del plano de descarte por lejanía.

El contenido de las luces se indicará más adelante en sus apartados específicos.

RENDERIZADO DE OBJETOS

Una vez cargados los objetos con las características adecuadas, es preciso organizar el dibujado de las entidades separado en las etapas descritas.

El encargado de esta tarea será un componente que herede de DrawableComponent. En su método de dibujado tendrá tres fases separadas.

La primera recorrerá todas las entidades visibles y llamará a su método de dibujado. Los modelos, por lo tanto, son los responsables de saber cómo deben dibujar su geometría a los g-buffers. Serán necesarios cuatro g-buffers: posición, normal, especular y color.

En la segunda fase se recorrerán todas las luces activas en la escena. Para cada una de ellas se dibujará por separado su aporte de luz difusa y especular. Esto es debido a que la luz difusa debe multiplicar su color por el de la superficie, mientras que la luz especular se añade sin cambios.

La forma exacta de calcular la intensidad se detallará en la siguiente sección. Es importante notar que si almacenamos los valores del color de la luz entre 0 y 1 (rango por defecto) puede saturarse y perder intensidad al modularse con el color difuso. Por lo tanto, dividiremos todos los valores calculados de intensidad en esta fase por la mitad y los doblaremos en la siguiente.

Finalmente, una tercera fase recoge los valores de iluminación y los combina con el g-buffer de color para formar la escena final.

ILUMINACIÓN

En esta sección detallaremos el modo de calcular el aporte de cada tipo de luz a la iluminación de la escena, explicando el modelo simplificado que se utilizará para cada una y las fórmulas utilizadas para la atenuación y la intensidad.

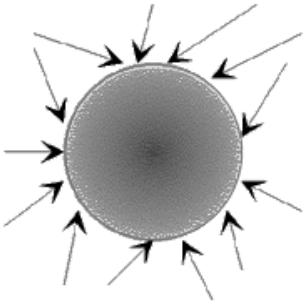
Las variables que intervendrán en las formulas son las siguientes:

- C (color): Valor de color de la luz.
- N (normal): Vector normal de la superficie a iluminar.
- V (view): Vector direccional desde la cámara al pixel.
- L (light): Vector de la dirección de la luz.
- H (half): Vector intermedio entre la dirección de la luz (L) y la dirección de la cámara (V).
- S (specular): Color del reflejo especular de la superficie.
- α: Coeficiente de brillo especular de la superficie.

Todas las luces deberán tener como mínimo la propiedad de color y leer su valor del archivo XML.

LUZ DE AMBIENTE

Este tipo de luz no proviene de una fuente concreta. Se utiliza para modelizar la luz resultante de la combinación de la luz dispersa de todas las fuentes después de rebotar en los objetos de la escena.



Existirá una única luz de ambiente en cada escena y se aplicará sobre todos los pixeles de la pantalla en una única pasada.

La aproximación que utilizaremos será la más sencilla dentro la las opciones dinámicas (es decir, no precalculadas con las luces de la escena), que es considerarla un aporte constante para toda la escena, aunque variable en tiempo real si la aplicación lo requiere. Este valor constante será igual a la propiedad de color de este tipo de luces y afectará únicamente a la luz difusa.

Por lo tanto, para este tipo de luces:

$$I_{diffuse} = C$$

$$I_{specular} = 0$$

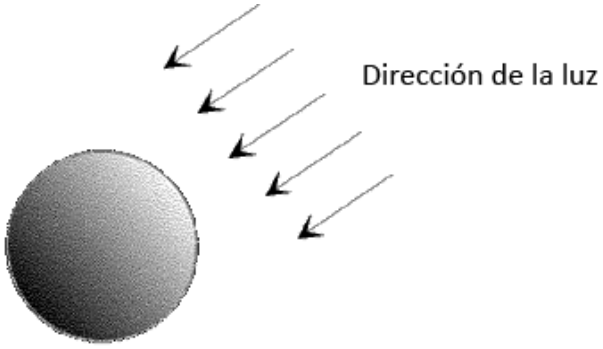
El resultado con esta aproximación es muy plano en aquellas zonas que no reciben iluminación directa. Las aproximaciones más modernas incluyes técnicas de *ambient occlusion* que dan volumen a la escena incluso donde no llega la luz. Discutiremos en el apartado de trabajos futuros la integración de este tipo de técnicas con esta arquitectura de render.

LUZ DIRECCIONAL

El tipo de luz más sencilla de implementar después de la de ambiente es la luz direccional. Estas luces provienen de fuentes de luz situadas a una distancia lo suficientemente lejana como para considerar sus rayos paralelos.

Se utiliza por ejemplo para simular la luz solar o para dar una iluminación base a la escena que marque los volúmenes principales.

Este tipo de luces deberán tener como propiedad la dirección de la luz y leerla del XML.



Con estas características, las luces direccionales afectarán a toda la escena con la misma intensidad, del mismo modo que ocurría con la luz de ambiente. Se dibujará por lo tanto como una pasada a toda la pantalla.

La diferencia fundamental con la luz de ambiente es que el aporte de iluminación se calcula en función del ángulo de incidencia de la luz al objeto, según la aproximación dada por las siguientes fórmulas:

$$I_{diffuse} = C * (N \cdot L)$$

$$I_{specular} = S * C * (N \cdot H)^{\alpha}$$

$$(para\ N \cdot L > 0 \quad y \quad N \cdot H > 0, \quad 0\ en\ caso\ contrario)$$

La luz difusa tiene en cuenta únicamente el color y el producto escalar del vector de dirección de la luz con la normal de la superficie, que determina el ángulo de incidencia. Sólo se tienen en cuenta aquellos pixeles donde el ángulo de incidencia tiene un valor positivo.

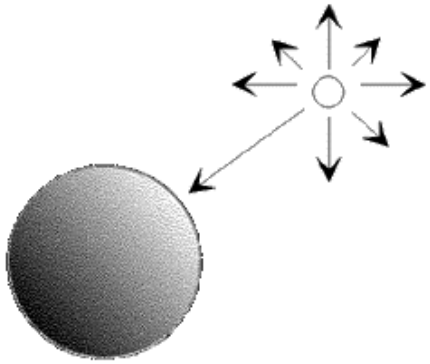
El cálculo de la luz especular es ligeramente más complicado. La aproximación utilizada se corresponde con el modelo de Blinn, que utiliza el vector intermedio entre la luz y la cámara para aproximar el reflejo especular de la luz. El exponente α utilizado se leerá de la textura de especular y determina cómo de concentrado está el brillo.

LUZ OMNIDIRECCIONAL

Con los dos tipos de luces explicados anteriormente no es posible iluminar adecuadamente una escena. Es necesario por tanto utilizar luces posicionales que permitan localizar las fuentes de luz cuando éstas son cercanas. La más sencilla de las luces posicionales es la omnidireccional, que afecta por igual en todas las direcciones. Un ejemplo de este tipo de luces puede ser la luz producida por una antorcha o por una bombilla.

La intensidad de la luz producida por este tipo de luces depende de la distancia, dejando de afectar a los objetos suficientemente alejados de esta.

Deberán, por tanto, poseer las propiedades de posición y atenuación (inicial y final) y ofrecer la posibilidad de cargarlas de los archivos de datos XML.



La forma más adecuada de modelizar este tipo de luces para renderizarlas es como una esfera de luz. El radio de la esfera está determinado por el valor del final de la atenuación, ya que a partir de este el valor de iluminación será siempre cero.

Las fórmulas para calcular la iluminación de este tipo de luces se puede obtener a partir de la formula anterior (I'), modulándola con la atenuación:

$$Att = smoothstep(Att_{initial}\ ,\ Att_{final},\ \ \|L\|\ )$$

$$I_{diffuse} = I'_{diffuse} * Att$$

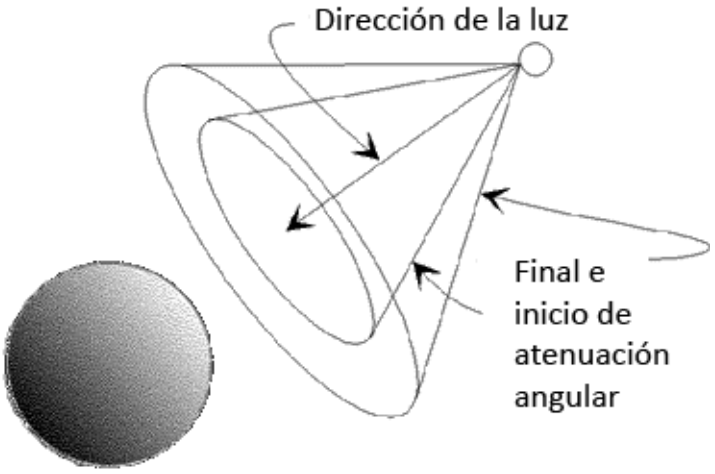
$$I_{specular} = I'_{specular} * Att$$

La atenuación de la luz debería utilizar un modelo matemático realista donde la intensidad disminuyera de forma aproximadamente inversamente proporcional al cuadrado de la distancia. No obstante, la práctica demuestra que es mucho más interesante poder especificar manualmente los radios de atenuación para facilitar la edición de las escenas. La interpolación entre estos valores se realizará con una función sigmoidal (smoothstep) q suaviza las transiciones. Por lo demás, la intensidad se calcula del mismo modo que para una luz direccional, con la particularidad de que el vector de luz (L) se obtiene desde el centro de la luz hasta el pixel iluminado, siendo su longitud utilizada para calcular el valor de la atenuación en el pixel.

LUZ DE FOCO

Utilizaremos también otro tipo de luz posicional además de la omnidireccional. Las luces de foco se utilizan para combinar las virtudes de las luces direccionales con las de las omnidireccionales. Además de su posición, tienen una dirección hacia la que están dirigidas y un ángulo máximo de iluminación. Como su propio nombre indica, un ejemplo muy claro de este tipo de luz son los focos utilizados en fotografía.

Este tipo de luces comparte con las omnidireccionales las propiedades de posición, atenuación inicial y atenuación final. Hereda también la propiedad de dirección de las luces direccionales. Junto con estas, añade dos nuevas propiedades, la atenuación angular inicial y final, especificados como valores entre 0 y 1, obtenidos como el seno del ángulo formado con el vector de dirección. Como el resto de luces, todas estas propiedades deberían ser modificables y editables desde XML.



Como hicimos con las luces omnidireccionales, obtendremos la intensidad de estas luces en función de la fórmula de las luces direccionales (I'):

$$Att = smoothstep\big(Att_{initial}\,,\, Att_{final},\ \|L\|\,\big)$$

$$AttAng = smoothstep(AttAng_{initial}\,,\, AttAng_{final},\ \sqrt{1-(L\cdot Dir)^2}\,)$$

$$I_{diffuse} = I'_{diffuse} * Att\ * AttAng$$

$$I_{specular} = I'_{specular} * Att\ * AttAng$$

El cálculo de atenuación por distancia se realiza exactamente del mismo modo que en las luces omnidireccionales, interpolando suavemente entre las dos distancias. A esta atenuación hay que multiplicarle la atenuación angular. Para ello interpolamos con la función sigmoidal entre las atenuaciones angulares, calculando el valor actual con el seno del vector de luz con la dirección de la luz. La forma más eficiente de obtener el valor del seno es en función del coseno, que a su vez se puede obtener como producto escalar entre ambos vectores normalizados.

GESTIÓN DE ESCENAS

Para facilitar la construcción de las escenas y la edición de las luces, crearemos ficheros XML que agrupen todos los elementos de la escena. El formato de estos archivos es propio y contiene las propiedades que hemos detallado de cada uno. La estructura viene determinada por las propiedades de las clases de contenido, puesto que se importará automáticamente.

```
<?xml version="1.0" encoding="utf-8"?>
<XnaContent xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics"
            xmlns:DeferredEngine="DeferredEngineContentPipelineExtension"
            xmlns:DeferredViewer="DeferredViewerContentPipelineExtension">
       <Asset Type="DeferredViewer:Scene3DContent">
              <Ambient>0.01 0.01 0.01</Ambient>
              <Lights>
                     <Light Type="DeferredEngine:DirectionalLightContent">
                            <Intensity>1.000000</Intensity>
                            <Color>0.7 0.6 0.2</Color>
                            <Direction>0.5 -1 -0</Direction>
                     </Light>
                     <Light Type="DeferredEngine:OmniLightContent">
                            <Intensity>0.500000</Intensity>
                            <Color>0.3 0.5 0.9</Color>
                            <Position>0 0 0</Position>
                            <AttStart>35.000000</AttStart>
                            <AttEnd>70.000000</AttEnd>
                     </Light>
                     <Light Type="DeferredEngine:SpotLightContent">
                            <Intensity>2.0000</Intensity>
                            <Color>1 1 1</Color>
                            <Position>200 300 100</Position>
                            <Direction>-2 -3 -1</Direction>
                            <AttStart>300.000000</AttStart>
                            <AttEnd>500.000000</AttEnd>
                            <AttAngStart>0.2</AttAngStart>
                            <AttAngEnd>0.4</AttAngEnd>
                     </Light>
              </Lights>
              <Cameras>
                     <Camera>
                            <Key>Camera01</Key>
                            <Value Type="DeferredEngine:LookAtCameraContent">
                                   <Position>0 100 500</Position>
                                   <Fov>0.785398</Fov>
                                   <NearPlane>10.0000</NearPlane>
                                   <FarPlane>1000.000000</FarPlane>
                                   <Target>0 0 0</Target>
                            </Value>
                     </Camera>
              </Cameras>
              <Entities>
                     <Entity Type="DeferredViewer:ObjectEntityContent">
                            <EntityName>Object01</EntityName>
                            <Model><Reference>#Teapot</Reference></Model>
                            <Position>0 0 0</Position>
                            <Orientation>0 0 0</Orientation>
                     </Entity>
              </Entities>
       </Asset>
       <ExternalReferences>
              <ExternalReference ID="#Teapot" TargetType="Graphics:NodeContent">
                     .\Objects\Teapot\teapot.fbx
              </ExternalReference>
       </ExternalReferences>
</XnaContent>
```

IMPLEMENTACIÓN

DIBUJADO DE LA ESCENA A LOS G-BUFFERS

Una vez están cargados todos los modelos que queremos utilizar, simplemente hay que dibujar todas las mallas del modelo usando su efecto.

Como no estamos utilizando el shader por defecto, será necesario especificar al efecto la matriz de transformación de los vértices y otros parámetros requeridos por el vertex shader que, en la arquitectura predeterminada, ya asigna internamente XNA. Los parámetros que deberemos pasar serán las matrices de transformación de coordenadas de vértice a vista y a pantalla, y la distancia máxima de dibujado de la cámara.

También será necesario indicar las texturas de color, normal y reflexión especular del modelo para el pixel shader.

VERTEX SHADER

El vertex shader es el encargado de preparar todos los datos y pasárselos al pixel shader. La información mínima que debe tener como salida es la posición del vértice transformado.

Como explicamos anteriormente, para calcular la posición el vértice tiene que ser transformado por las matrices de mundo, vista y proyección. Al tratarse de transformaciones lineales, la matriz de transformación se calculará una vez en la aplicación como combinación lineal de las tres y se pasará como una única matriz al shader.

Además de la posición, nosotros necesitaremos pasar más datos al pixel shader. Puesto que el objeto estará texturizado necesitaremos conocer para cada vértice sus coordenadas de textura, es decir, en un rango [0, 1], que posición de la textura se debe tomar para el dibujado. Estas coordenadas de textura 2D se llaman normalmente coordenadas UV de textura y están almacenadas en la información del modelo, así que no hace falta más tratamiento que pasarlo directamente al pixel shader.

Como veremos más adelante, en el g-buffer que utilizaremos para almacenar la posición lo haremos en coordenadas de vista, por lo que es necesario pasarle al pixel shader la posición del vértice transformado por la matriz adecuada. Al igual que con la posición proyectada, sólo se calculará una vez en la aplicación y luego se pasará al vertex shader. Para acotar los valores de posición entre 0 y 1, lo dividiremos entre la distancia máxima de dibujado (FarPlane).

Finalmente, deberemos calcular la matriz de espacio tangente, necesaria para el cálculo de la normal usando texturas de normales. Si no usáramos este tipo de texturas, sería necesario únicamente transferir al pixel shader la información de la normal en espacio de vista, pero en nuestro caso tendremos que acompañar esta información con el vector tangente y binormal de la superficie en el vértice, todo esto también en espacio de vista, formando una matriz de 3x3.

```
// Estructura de salida del vertex shader
struct VS_OUTPUT_TANGENT
{
      float4 Position         : POSITION;
      float2 UV               : TEXCOORD0;
      float4 PosView          : TEXCOORD1;
      float3x3 TangentToWorld : TEXCOORD2;
};

// Vertex shader
VS_OUTPUT_TANGENT TangentVS(VS_INPUT_TANGENT In)
{
      VS_OUTPUT_TANGENT Out;

      // Posición proyectada
      Out.Position = mul(In.Position, WorldViewProj);

      // Coordenadas de textura
      Out.UV = In.UV;

      // Posición en vista
      Out.PosView = mul(In.Position, WorldView);

      // Matrix de espacio tangente
      float3 normal = mul(In.Normal, (float3x3)WorldView);
      float3 tangent = mul(In.Tangent, (float3x3)WorldView);
      float3 binormal = mul(-In.Binormal, (float3x3)WorldView);
      Out.TangentToWorld = float3x3(tangent, binormal, normal);

      return Out;
}
```

PIXEL SHADER

La información de la estructura de salida del vertex shader llega interpolada linealmente como input al pixel shader, excepto el campo de posición que se utiliza automáticamente para decidir la posición del pixel.

Con esta información, debemos calcular el color de cada uno de los cuatro g-buffers que vamos a utilizar. A continuación describiremos la obtención de cada uno, acompañándolo de una imagen con el color que tiene en un ejemplo donde renderizamos una tetera (modelo clásico de pruebas en 3D) verde de plástico.

```
//Estructura de salida del pixel shader
struct PS_OUTPUT
{
      float4 Color:    COLOR0;
      float4 Depth:    COLOR1;
      float4 Normal:   COLOR2;
      float4 Specular: COLOR3;
};
```

COLOR

Ilustración 7. G-buffer de color

Más apropiadamente llamado color difuso o albedo, es el color natural de la superficie cuando está completamente iluminada. Lo obtendremos simplemente leyendo el valor de la textura en las coordenadas de mapeado del pixel, interpoladas del valor que pasamos antes desde el vértice.

```
Out.Color = tex2D(DiffuseSampler, In.UV);
```

Aunque en el ejemplo tiene un color plano por simplificar, lo normal es que tenga una textura con el color del archivo de imagen pero transformada según el mapeado del objeto.

POSICIÓN

Ilustración 8. G-buffer de posición

Esta información se necesita para las luces posicionales (luces omnidireccionales y luces de foco) que afectan a los objetos en función de la distancia a la que se encuentran. Las luces globales (luces de ambiente y luces direccionales) afectan a todos los objetos por igual y no requieren esta información.

Para ahorrar cálculos, nos aprovecharemos de la circunstancia de que las coordenadas X e Y están implícitas en la posición 2D del pixel y almacenaremos únicamente el valor de profundidad. Para almacenar el valor de profundidad en el rango [0, 1], lo dividiremos entre la distancia máxima. Además, será necesario cambiar el signo ya que el eje Z es negativo hacia dentro de la pantalla.

```
Out.Depth = -In.PosView.z / FarPlane;
```

En la imagen podemos apreciar como los valores de los bordes (que tienen una mayor profundidad) tienen un color ligeramente más claro.

NORMAL

Ilustración 9. G-buffer de normal

El vector normal de la superficie es necesario para calcular el ángulo de incidencia de la luz sobre ésta. Para almacenar estos vectores como valores de color, será necesaria una transformación, puesto que el vector (normalizado) está en el rango [-1, 1] y debe almacenarse en el rango [0,1].

Para el cálculo de la normal, primero debemos obtener el valor de la textura de normales y ponerlo de nuevo en el rango [-1, 1] para poder trabajar con él. Después, hemos de transformar el vector con la matriz de espacio tangente recibida del vertex shader. Esto es así porque los valores almacenados en la textura son locales a la superficie, sin tener en cuenta la orientación que pueda tener ésta. Al transformarlo, el vértice ya se encuentra en espacio de vista. Aunque el vector de la textura está normalizado, es recomendable volver a normalizarlo para corregir el efecto producido por la interpolación lineal sobre la matriz de espacio tangente. Tras esto, bastará con volver a ponerlo en el rango [0, 1]. El alpha de este g-buffer no se utilizará para nada.

```
float3 normal = tex2D(NormalSampler, In.UV) * 2 - 1;
normal = mul(normal, In.TangentToWorld);
Out.Normal.xyz = normalize(normal) * 0.5 + 0.5;
Out.Normal.a = 1;
```

Se puede observar en la imagen que las partes de la superficie que están más orientadas hacia la derecha tienen mayor componente rojo (X), las que están más orientadas hacia arriba tienen una mayor componente verde (Y) y las que son más frontales tienen mayor cantidad de azul (Z).

ESPECULAR

Ilustración 10. G-buffer de especular

Del mismo modo que ocurría con el g-buffer de color, no es necesario transformar de ninguna forma la textura, que tendrá ya el formato adecuado en la imagen.

```
Out.Specular = tex2D(SpecularSampler, In.UV);
```

Puede sorprender el contenido de la imagen, sobre todo teniendo en cuenta dijimos que la tetera del ejemplo era de color verde. Lo cierto es que la luz que se refleja especularmente normalmente no adquiere el color de la superficie. Algunos materiales, como los metales o algunos tejidos como el terciopelo sí que tintan esta luz y no necesariamente del mismo color que la luz difusa, por lo que es necesario ofrecer la posibilidad de cambiar este color. En nuestra tetera de plástico el reflejo será de color blanco.

En la imagen no se ve el contenido del canal alpha, pero sí se almacena en el g-buffer. Es importante, puesto que el valor almacenado (en escala de grises) es el exponente utilizado para calcular la componente de luz especular.

FASE DE ILUMINACIÓN

Tras rellenar el contenido de los g-buffers con la información de la escena, ya es posible pasar a la fase de iluminación.

En esta fase utilizaremos el g-buffer de posición y el de normales, y el canal alpha del g-buffer de especular. No necesitaremos aun el de color y el color de especular porque vamos a renderizar la luz difusa y especular en dos rendertargets separados. Lo haremos así porque hemos pospuesto la composición de la escena, y no podemos mezclar los dos tipos de luz porque una se ve afectada por el color difuso y otra por el color especular del material.

Para la luz de ambiente (recordemos que es única) no es necesario renderizar nada. Nos aprovecharemos de la particularidad de que afecta a toda la escena por igual e inicializaremos el rendertarget de luz difusa con el color de esta luz. Mientras, la inicialización del rendertarget de luz especular la haremos en color negro.

Para el resto de luces, utilizaremos un shader específico para calcular los valores de la iluminación.

VERTEX SHADER

Las luces direccionales no necesitarán de vertex shader, ya que al igual que la luz de ambiente afectan a todos los pixeles.

Sin embargo, para las luces omnidireccionales y de foco si hemos utilizado un shader específico, aunque bastante sencillo. La atenuación de estas luces ya haría que sólo se iluminaran los pixeles adecuados, pero al utilizar un modelo específico nos ahorramos procesar muchos pixeles.

Para las luces omnidireccionales partiremos de una esfera de radio 1 y la escalaremos según la distancia máxima de atenuación. Más allá de esta esfera ningún pixel se verá afectado.

```
VS_OUTPUT SimpleVS(float4 position: POSITION)
{
	VS_OUTPUT Out;

	position.xyz *= LightAttEnd;
	Out.Position = mul(position, WorldViewProj);
	Out.Position2 = Out.Position;

	return Out;
}
```

Podemos observar que la posición se almacena 2 veces en el shader, con el mismo valor. El motivo es que el valor de salida de posición se utiliza para calcular la posición de los pixels, pero no se les pasa a estos como entrada. Si queremos que el pixel sepa cuál es su posición (y la necesitará para recomponer la posición de la superficie) tendremos que pasársela explícitamente con otro parámetro.

Si para las luces omnidireccionales partíamos de una esfera, para las luces de foco utilizaremos una semiesfera, también de radio 1. Aunque intuitivamente es fácil asociar las luces de foco con la figura de un cono, esto no es del todo válido para la base, ya que en el cono el corte con la base es plano en lugar de curvo como debería ser en este tipo de luces.

Consideraremos la semiesfera como una luz de foco de 90° de atenuación angular final. Para todos los vértices que no sean el que está en el origen los escalamos por su atenuación angular final, que recordemos es equivalente a su proyección en el eje perpendicular a la dirección, es decir, perpendicular al eje Z en la semiesfera base. Con la esfera achatada en XY según la atenuación angular, ajustamos la coordenada Z para mantener el radio 1 original.

Tras este primer paso obtenemos el cono de luz que buscábamos, en su forma unitaria. En este momento le aplicamos la misma transformación que a la esfera de las luces omnidireccionales, escalándola con la distancia de atenuación y transformándolo por la matriz combinada de mundo-vista-proyección.

```
VS_OUTPUT SpotVS(float4 position: POSITION)
{
      VS_OUTPUT Out;

      if(length(position.xyz) > 0.01)
      {
            position.xy *= LightAttAngEnd;
            position.z = -sqrt(1 - (position.x * position.x +
                                    position.y * position.y));
      }

      position.xyz *= LightAttEnd;
      Out.Position = mul(position, WorldViewProj);
      Out.Position2 = Out.Position;
      return Out;
}
```

Al igual que con las luces omnidireccionales, almacenamos también un segundo parámetro de posición que le llegará al pixel shader.

PIXEL SHADER

Cada tipo de luz (direccional, omnidireccional y de foco) tiene su pixel shader específico, pero hay partes comunes en todos ellos.

Para empezar, el cálculo de la iluminación es igual en las tres, por lo que lo unificaremos en una única función.

```
// Lighting function
// n: normal vector
// v: view vector
// l: light vector
// s: specular coeff
// i: light intensity
float3 CalculateLighting(float3 n, float3 v, float3 l, float s, float i)
{
      float3 h = normalize(l + v);

      float3 light = 0;
      light.y = saturate(dot(n, l)) * i * 0.5;
      light.z = light.y * pow(dot(n, h), 1.0f / s);
      return light;
}
```

La función de iluminación implementa las fórmulas detalladas en el apartado de diseño utilizando los distintos parámetros recibidos (vectores, coeficiente de especular e intensidad). La fórmula original no contempla el parámetro de intensidad, pero se ha incluido finalmente como un multiplicador del color para facilitar la animación de la intensidad variando un único valor en lugar de los 3 componentes del color.

Otra parte común es el cálculo de la posición. A partir de la posición calculada en el vertex shader tenemos que calcular dos cosas: las coordenadas de textura y la posición de la superficie del objeto.

Las coordenadas de textura las podemos obtener cambiando de rango la posición, del rango [-1, 1] en el que se recibe la posición al rango [0, 1] de las texturas, teniendo en cuenta que el eje y está invertido. Además, para mejorar la calidad del resultado, será necesario añadir medio pixel para samplear justo en medio del pixel y evitar el filtrado de la textura.

```
float2 texCoord = (Position.xy+1) * float2(0.5,-0.5) + 0.5 / Resolution;
```

Para el cálculo de la posición del objeto disponemos de un vector que hemos pasado desde la aplicación que, multiplicado por la posición del pixel (que establecemos en el plano Z=1) y la profundidad leída del g-buffer nos dará la posición en 3D del pixel de la superficie.

```
pos=float3(Position.xy,1)*QuadToView*tex2D(PositionSampler, texCoord).r;
```

Una vez tenemos la posición de la superficie, podemos calcular los vectores de vista y de luz, restandola a la posición de la cámara y de la fuente de luz (en el caso de las luces omnidireccionales y de foco) respectivamente. Las posiciones de estos objetos son comunes para todos los pixeles y serán pasados desde la aplicación.

Para poder calcular la iluminación nos queda entonces el vector normal, el coeficiente de especular y la intensidad y color de la luz. Los dos primeros los podemos obtener de los g-buffers, recordando cambiar de rango otra vez el vector de la normal. La intensidad y el color de la luz, por otra parte, vienen dados por la aplicación.

```
float4 specular = tex2D(SpecularSampler, texCoord);

float3 n = normalize(tex2D(NormalSampler, texCoord).rgb * 2 - 1);
```

Dependiendo de cada pixel shader, puede ser necesario también calcular la atenuación, siguiendo las fórmulas que se han diseñado. Una vez obtenidas, se utilizarán para modular el valor de la intensidad.

Con esto ya disponemos de todos los datos para llamar a la función de iluminación que detallamos anteriormente. Su resultado son los valores de intensidad de luz difusa y especular en el punto que, multiplicados por el color, nos dan los valores de salida para cada uno de los rendertargets.

```
float3 light = CalculateLighting(n, v, l, specular.a, i);

Out.Diffuse = float4(Color * light.y, 1);
Out.Specular = float4(Color * light.z, 1);
```

Para ilustrar mejor el resultado de esta fase, mostraremos dos imágenes con el resultado de iluminar la escena de ejemplo de la tetera descompuesta en g-buffers, utilizando un par de luces, una de foco y otra omnidireccional como contraluz para resaltar el perfil.

En las imágenes se puede ver como la mayor parte de la luz se concentra en el rendertarget de luz difusa, mientras que solo unos mínimos brillos se muestran en el rendertarget de luz especular. Ambos tipos de luz son necesarios y tienen distintos propósitos ya que, mientras la luz difusa nos muestra la información de color y forma de la escena, la luz especular añade información visual sobre el tipo de superficie de los objetos.

Otro detalle a tener en cuenta es el de que, aunque los rendertargets que vemos están en escala de grises, podrían contener color perfectamente si se hubiera utilizado una luz coloreada.

Ilustración 11. Iluminación difusa

Ilustración 12. Iluminación especular

COMPOSICIÓN DE LA ESCENA FINAL

La última pasada de render es la más sencilla. Recibe como entradas las dos texturas con la iluminación calculada y los g-buffers de color y color de especular.

El pixel shader multiplica cada textura de iluminación por su color correspondiente y las suma para obtener el resultado final.

Como añadido, hemos utilizado el canal alpha de la textura de color difuso para permitir almacenar auto-iluminación a algunos objetos. Un valor de alpha de 1 (por defecto) no tendrá auto-iluminación, mientras que si la textura especifica un valor menor se añadirá su complementario a la luz difusa.

```
float4 color = tex2D(ColorSampler, texCoord);
float4 specular = tex2D(SpecularSampler, texCoord);
float autoilum = 1 - color.a;

float4 light = tex2D(ScreenSampler, texCoord) * 2;
float4 slight = tex2D(SpecularLightingSampler, texCoord) * 2;

return color * (light * color.a + autoilum) + specular * slight;
```

Finalmente, siguiendo con el ejemplo anterior, este sería el resultado para nuestra tetera verde.

Ilustración 13. Resultado final del ejemplo

RESULTADOS OBTENIDOS

Para concluir mostraremos por separado las distintas características del motor, de forma que se puedan observar las posibles opciones en la creación de escenas.

Empezaremos diseccionando los tipos de luces y las propiedades del material sobre la tetera de ejemplo y, finalmente, haremos un resumen de las etapas sobre un modelo más complejo.

LUCES

Examinaremos primero el efecto de cada uno de los tipos de luces sobre una tetera de color gris, observando además el número de pixeles afectados.

LUZ DE AMBIENTE

Ilustración 14. Resultados: Luz de ambiente

En la prueba el color de ambiente era de un gris oscuro al 10%, un aporte adecuado para una escena no muy iluminada. Es importante controlar la cantidad de luz ambiente, puesto que, como consecuencia de afectar por igual a todos los pixeles, aplana mucho la escena.

En la imagen obtenida se observa claramente como aparece únicamente la silueta de la tetera. Si el color de ambiente fuera mucho más alto, blanco por ejemplo, sobresaturaría la luz difusa y obtendríamos un resultado plano del color de la textura de color difuso independientemente del número y tipo de luces utilizadas, sólo aclarado en las zonas afectadas por la luz especular. Se recomienda su uso sólo para eliminar el color negro en las zonas completamente en sombra.

Recordemos que, al hacer uso del relleno de inicialización, no se calcula nada en el pixel shader.

LUZ DIRECCIONAL

Ilustración 15. Resultados: Luz direccional

La siguiente prueba la realizamos con una luz direccional cuya dirección proviene desde arriba a la derecha. La intensidad de la luz es de 0’5 y el color es blanco. Estos parámetros de luz los repetiremos en los otros modelos de luces para poder comparar adecuadamente.

Con este tipo de luz ya si se observa perfectamente el volumen de la tetera, no solo la silueta. Se ha mantenido la luz ambiente para poder entender bien la figura, ya que de no ser así las zonas orientadas en la misma dirección que la luz (hacia abajo izquierda) serían completamente negras.

Aunque el pixel shader utilizado por este tipo de luces es ligeramente menos costoso por no tener que calcular la atenuación, es por lo general el tipo de luz más costoso puesto que afecta a todos los pixeles de la imagen incluso aunque no les aporte nada de iluminación.

LUZ OMNIDIRECCIONAL

Ilustración 16. Resultados: Luz omnidireccional

Aunque hemos visto alguna luz omnidireccional en los ejemplos anteriores, conviene observar el resultado de una de estas luces aisladas.

Uno de los aspectos más importantes de este tipo de luces es ajustar bien los valores de atenuación. Primero es necesario establecer el radio máximo para que afecte solamente a los objetos que queremos iluminar. Hay que ser cuidadoso porque al ser luces que van en todas direcciones son más difíciles de controlar. Después, es importante también elegir adecuadamente el punto del comienzo de la atenuación. Lo ideal es conseguir un equilibrio entre un núcleo intenso de luz (valores altos de atenuación) y un gradiente suave de transición (valores pequeños de atenuación).

Si renderizamos por separado la esfera que limita el efecto de la luz, podemos observar que el número de pixeles afectados es menos de la mitad de la pantalla, ahorrando tiempo de pintado.

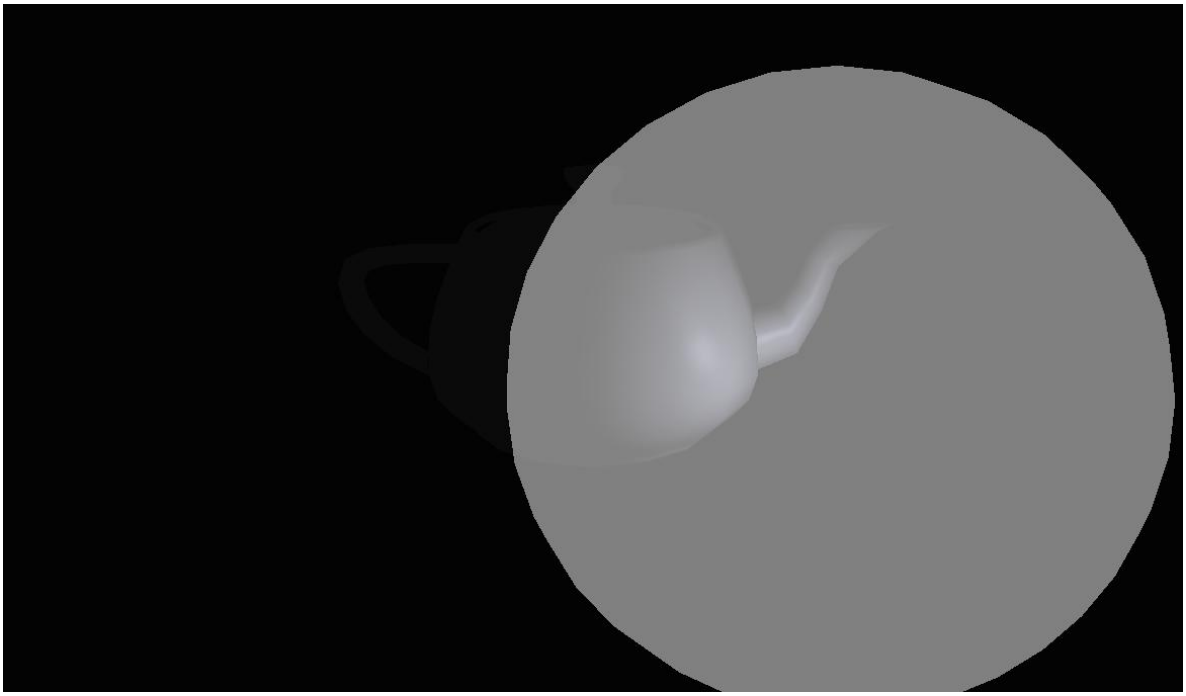


Ilustración 17. Resultados: Luz omnidireccional - límites

Podemos observar que el número de pixeles afectados es mucho menor que en las luces direccionales, que pintaban todos los de la pantalla.

La consecuencia más importante de limitar el procesado de pixeles de esta forma es que el coste de una luz es proporcional a su tamaño en pantalla. Esto supone una gran mejora con respecto al modelo anterior donde todas las luces tenían el mismo coste pero había un número limitado.

Por ejemplo, si tenemos un pasillo iluminado a base de multitud de candelabros de pequeño tamaño será posible asignar una luz de pequeño tamaño a cada uno de ellos, mientras que con la forma tradicional de renderizar los modelos habría sido necesario trucar la escena unificando algunas luces y afectando al realismo del resultado.

LUZ DE FOCO

Ilustración 18. Resultados: Luz de foco

El último tipo de luces que tenemos que probar es la luz de foco.

En la práctica se comportan exactamente como las omnidireccionales, pero tienen la capacidad de limitar su rango, lo que en algunos casos las hace más cómodas de usar.

Si con las luces omnidireccionales era importante ajustar bien los valores de atenuación, en este tipo de luces además de la atenuación por distancia hay que ajustar la atenuación angular. Por lo general no suele ser un problema, ya que el hecho de estar recurriendo a una luz de este tipo suele estar provocado por alguna circunstancia que requiere limitar el radio de acción, por lo tanto el ángulo máximo es fácil de ajustar. El ángulo de inicio de atenuación tiene las mismas consideraciones que la atenuación por distancia.

Dibujamos también con este tipo de luz el área de la pantalla al que afectará.

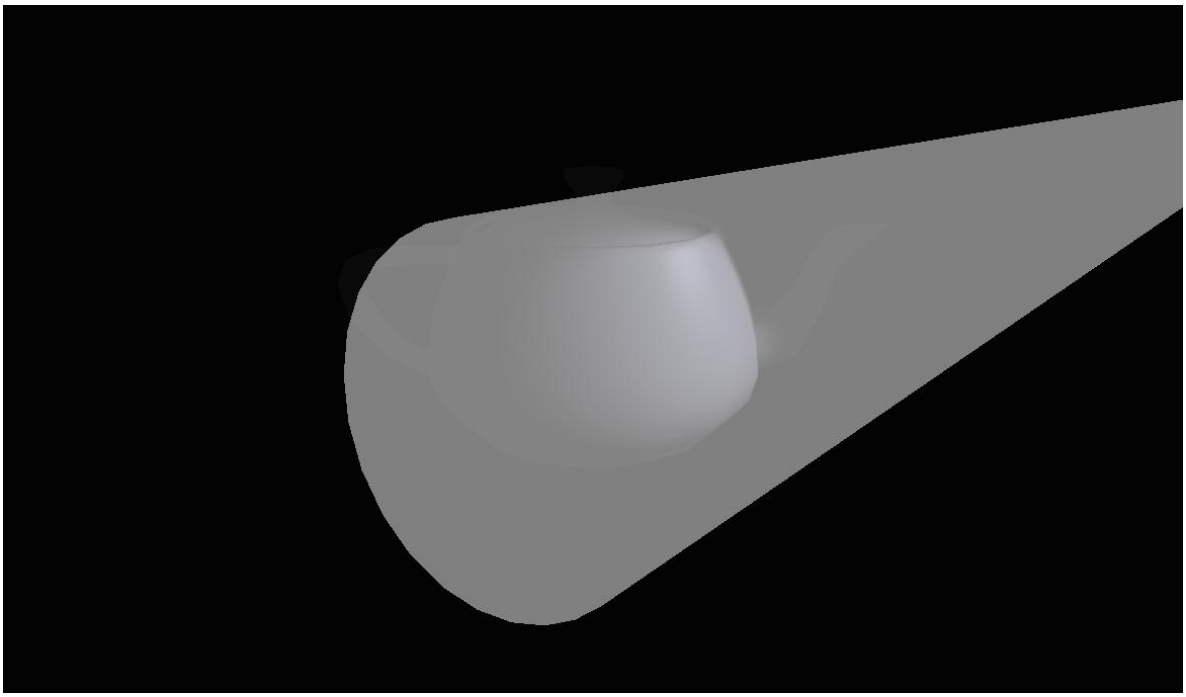


Ilustración 19. Resultados: Luz de foco - límites

La zona afectada por esta luz comprende un cono (estrecho, porque la atenuación angular final era baja) que no llega mucho más allá de la tetera, ahorrando pixeles que no tienen por qué ser dibujados.

Vemos con este cono también lo difícil que es apreciar la aportación de una luz, puesto que la parte superior de la tetera queda sin iluminar por la atenuación angular y no por el ángulo de incidencia como podría parecer en la primera imagen.

Dado que, en igualdad de atenuación por distancia, este tipo de luz es un subconjunto dentro del rango de luz que daría una luz omnidireccional, la zona afectada en este tipo de luces será de un menor número de pixeles.

MULTIPLES LUCES COLOREADAS

Uno de los requisitos que establecíamos explícitamente era que las luces podían cambiar su color. En las pruebas anteriores hemos utilizado siempre el color blanco para no desviar la atención del concepto que estuviéramos mostrando, pero también es necesario probar la funcionalidad del color de las luces.

Para probar que las luces pueden tener color y que el color se mezcla adecuadamente, hemos preparado una prueba con tres luces de foco separadas por la misma distancia y apuntando hacia un mismo punto localizado en una pared blanca. Cada una de estas luces tendrá un color primario puro. Existirán, pues, una luz roja, otra verde y otra azul.

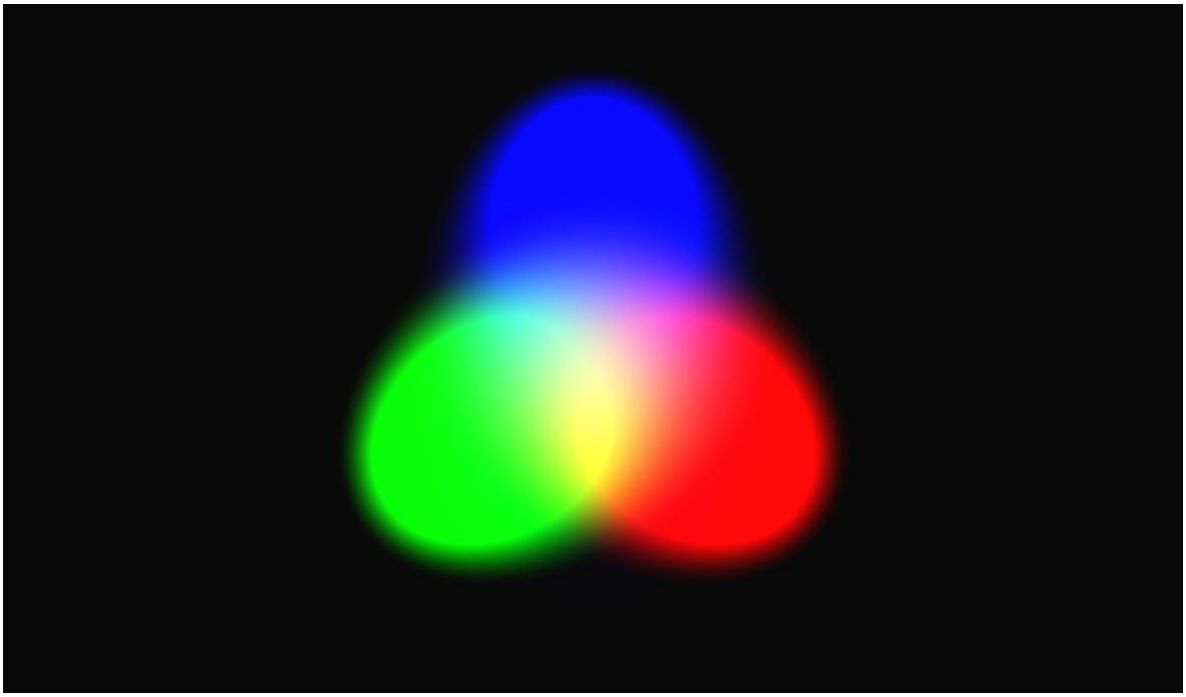


Ilustración 20. Resultados: Luces coloreadas

El resultado obtenido es el clásico dibujo con las distintas combinaciones de colores proyectado sobre la pared.

Comprobamos que, tal y como esperábamos, el color azul al mezclarse con el verde produce un color cyan, el verde con el rojo un color amarillo y el azul con el rojo un color magenta. La combinación de los 3 colores en el centro suma una luz blanca, como debe ser.

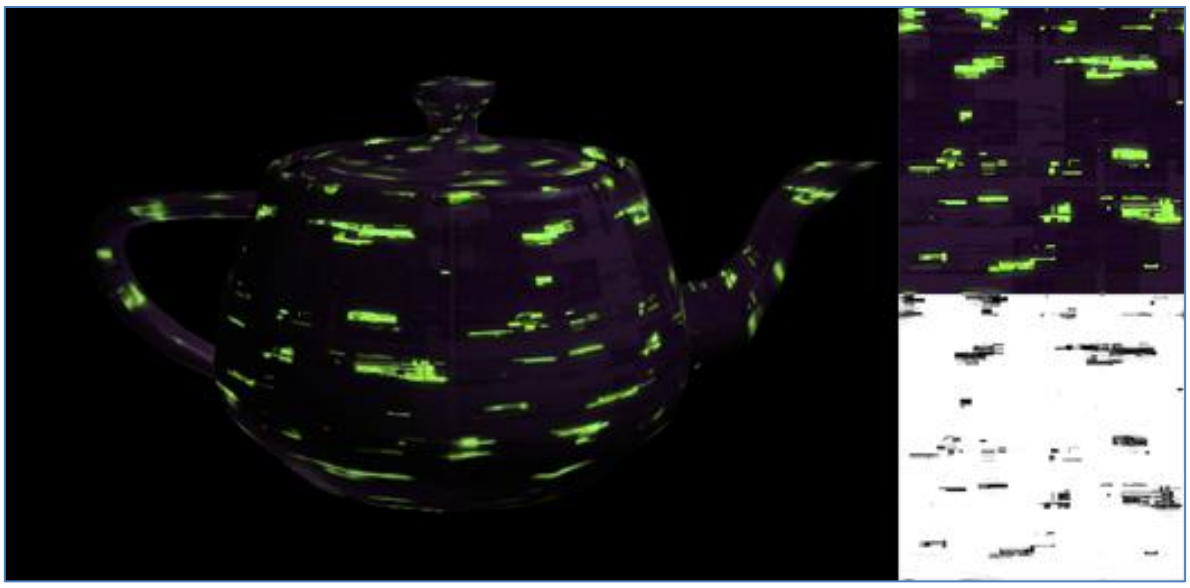
TEXTURAS

TEXTURA DE COLOR

Ya hemos visto anteriormente el resultado de una tetera con color plano, pero es posible utilizar cualquier textura. Para comprobarlo cargaremos una textura de bloques de piedra en la misma tetera, utilizando textura solamente en el canal de color.

En la imagen se muestra el resultado junto con la textura original, en la parte de arriba los canales RGB y en la parte de abajo el canal alpha.

Aprovecharemos también para mostrar un ejemplo de textura de color que usa el canal alpha como auto-iluminación (la iluminación de la escena se mantiene igual que en el ejemplo anterior).



TEXTURA DE NORMAL

Siguiendo con el ejemplo de los bloques de piedra, utilizaremos la textura correspondiente a las normales de los bloques, pero para observar mejor el efecto utilizaremos una textura de color gris plano.

Solo con la iluminación ya se consigue un resultado más realista que con la textura de color, donde se veía muy plano en las zonas con menos luz.

Una ventaja más de marcar volúmenes en la textura de normal es que el resultado cambia dinámicamente con la luz, por lo que es especialmente útil en objetos móviles o en escenas con luces que cambian de posición.

En cualquier caso, es importante cuidar la resolución y el nivel de detalle de estas texturas puesto que un exceso de detalle tiene un efecto contraproducente, añadiendo zonas de ruido que rompen el realismo de la escena.

Lo ideal es utilizar ambas texturas siempre que sea posible. Es tarea del artista determinar que detalles incluir únicamente en el color y cuales marcar además en la textura de normales para conseguir un resultado óptimo.

TEXTURA DE ESPECULAR

Quizás el tipo de textura menos intuitiva sea la de reflexión especular. Por lo tanto, vamos a poner tres ejemplos de distintos tipos de brillos que podemos conseguir con esta textura.

El ejemplo más sencillo es el de un material mate. Usaremos una tetera roja para distinguir entre el color difuso del color especular, que en nuestro caso será blanco como suele serlo en la mayoría de materiales.

En el canal alpha (representado en el cuadro de la esquina inferior derecha) hemos utilizado un valor alto, consiguiendo que la reflexión especular se disperse más para dar el aspecto mate.

Aunque coloquialmente se habla de los objetos de brillo mate como objetos sin brillo, esto no es del todo cierto, es simplemente que el brillo está menos concentrado. Si suprimiéramos por completo el efecto de la reflexión especular el resultado sería poco natural, ya que aunque no seamos conscientes de ello el ojo espera encontrar ese reflejo.

En contraposición a este ejemplo, modificaremos el coeficiente de reflexión especular para conseguir un objeto brillante con aspecto plástico.

Como vemos en la esquina inferior derecha, el valor utilizado en el canal alpha es bastante más oscuro. La consecuencia directa de esto es un brillo mucho más intenso y concentrado alrededor de un punto (que depende del ángulo de visión).

Modificando los canales RGB de la textura podemos conseguir también otros efectos. Por ejemplo, los objetos metálicos tienen generalmente reflexiones especulares coloreadas.

Si en este ejemplo no hubiéramos tintado el color del brillo el objeto no parecería metálico pese al color difuso dorado.

Este modelo de reflexión permite conseguir muchos resultados diferentes, pero no todos los materiales pueden representarse de esta forma. Por ejemplo, el reflejo anisotrópico de los materiales cromados es imposible de obtener de esta forma.

ESCENA COMPLEJA - SECUENCIA

Ilustración 21. Escena final

Para concluir la demostración de las posibilidades de nuestro sistema, observaremos su comportamiento con una escena más compleja en la que intervienen varios modelos y una gran cantidad de luces.

Además de nuestra clásica tetera, utilizaremos un modelo de acabado profesional con texturas detalladas y una habitación sencilla que sirva como entorno.

Lo primero que debería llamar la atención de esta escena es que el objeto del fondo se ve afectado por hasta 8 luces, algo que según que hardware se esté usando puede no ser posible usando la arquitectura tradicional de XNA y que, en cualquier caso, sería mucho más costoso de lo que es utilizando esta arquitectura.

En los siguientes apartados haremos un recorrido rápido por las dos etapas del render, viendo el estado de los g-buffers al acabar la primera etapa y el resultado de la iluminación difusa y especular al acabar la segunda.

El resultado final después de combinar las dos iluminaciones se puede ver en la ilustración 21.

G-BUFFERS

Ilustración 22. Escena final: G-Buffer color

En la ilustración 22 podemos ver el g-buffer que almacena la información de color. Se aprecia perfectamente cómo están texturizados los objetos, pudiendo apreciar los detalles como por ejemplo en la cara de Marilyn o en la superficie del lanzagranadas.

Llama la atención que parte de la iluminación se marca ligeramente en la textura de difuso, como por ejemplo aquellas zonas que por su morfología se encuentran más ocultas a la luz, que aparecen pintadas oscuras. Al hacer esto se ayuda a conseguir el efecto realista que no puede conseguirse aún con la iluminación en tiempo real, pero no se puede abusar de ello porque la iluminación pintada en la textura no es dinámica.

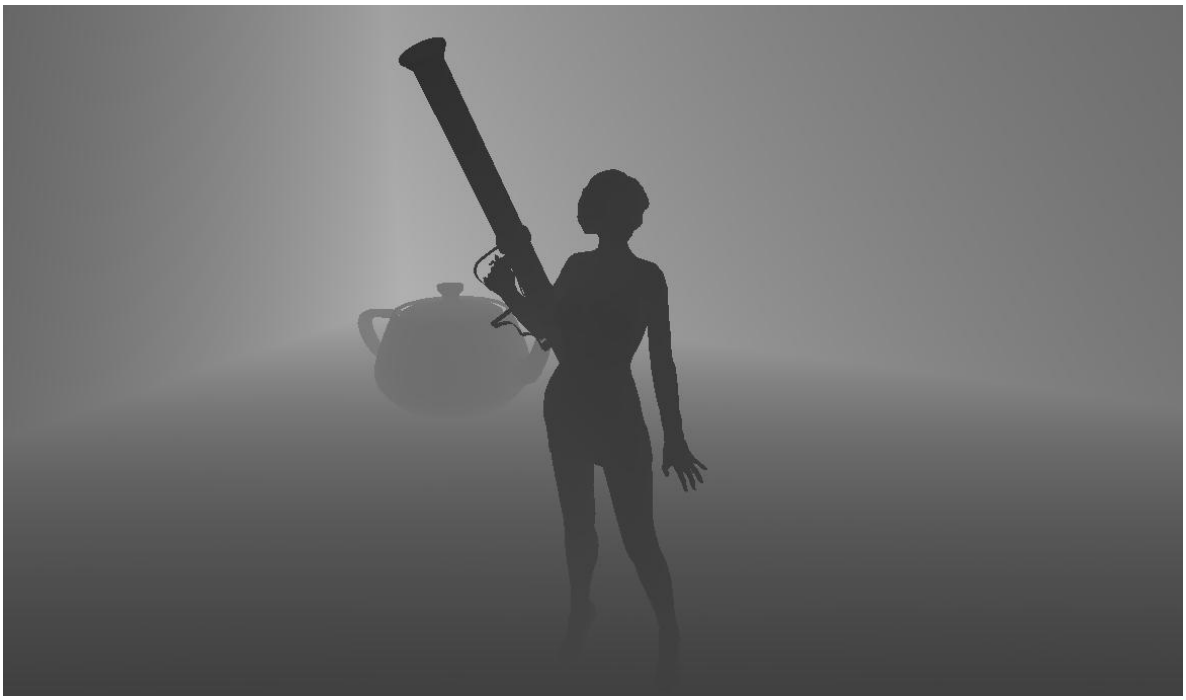


Ilustración 23. Escena final: G-Buffer posición

La ilustración 23 muestra el g-buffer de profundidad. Podemos ver más claramente que en el ejemplo previo cómo los pixeles se aclaran más cuanto más alejados de la cámara están.

Este es uno de los g-buffers más importantes, porque proporciona información muy clara sobre el volumen de la escena. Además de para reconstruir la posición 3D a partir de la posición 2D del pixel y su profundidad, se usa en muchas técnicas de post-procesado como por ejemplo para la detección de bordes.

Ilustración 24. Escena final: G-Buffer normal

El g-buffer de normales de la ilustración 24 muestra claramente la cantidad de detalle que aporta este tipo de texturas. Las líneas de los bloques de piedra de la tetera, las arrugas del vestido de Marilyn, el pelo, las clavículas, etc. son detalles que no están modelados con polígonos.

La ventaja de este tipo de texturas frente a la iluminación dibujada en la textura de color que veíamos en la ilustración 22 es que todo el detalle de estas texturas contribuye a la iluminación dinámica. Si pudiéramos ver una luz en movimiento subiendo y bajando frente a Marilyn, veríamos cómo las arrugas de su vestido se iluminan adecuadamente como si estuvieran modeladas.

Si aisláramos los canales RGB en imágenes en escala de grises, podríamos comprobar la equivalencia existente con los ejes XYZ. Los pixeles tienen mayor componente de rojo en las zonas orientadas en el eje positivo de las X, por ejemplo en el lado derecho de las piernas de Marilyn. Lo mismo ocurre con los otros dos ejes. La diferencia se puede observar muy claramente en los tres planos de la habitación, donde predominan el rojo, el verde o el azul según la dirección de su normal.

El motivo por el que no vemos unos tonos saturados puros es porque los vectores están normalizados y desplazados en el rango [0, 1]. Así, un vector completamente orientado hacia la cámara tendrá un valor de 1 en su eje Z (color azul), y sus componentes X e Y tendrán un valor 0, que corresponde con un valor de color 0’5, puesto que el color 0 se utiliza para el valor de coordenadas -1.

Ilustración 25. Escena final: G-Buffer especular

La información del g-buffer de reflexión especular se muestra en la ilustración 25. Destaca la presencia de color en esta imagen, puesto que como explicamos con anterioridad la mayoría de las reflexiones de los materiales no están tintadas de color, como ocurre con la habitación o la tetera.

En el modelo de Marilyn el artista ha pintado de rojo el vestido y los zapatos para conseguir un efecto de brillo aterciopelado. El lanzagranadas, por su parte, es metálico y ya explicamos que este tipo de materiales se caracterizan por el reflejo de color. Eso mismo ha querido conseguir el artista con el pelo, pintando el especular de color amarillo para conseguir reflejos dorados. Finalmente sorprende la textura de la piel, que aunque está muy cerca de los tonos grises tiene un ligero color azulado. Este es un recurso fruto de la práctica, puesto que el color azulado ayuda a equilibrar el tono anaranjado que tiene normalmente la textura de piel, dándole un aspecto más realista.

La tetera no se muestra tampoco con un color plano, ya que se aprovecha para marcar las zonas que están más pulidas y que por tanto tienen una mayor intensidad de brillo.

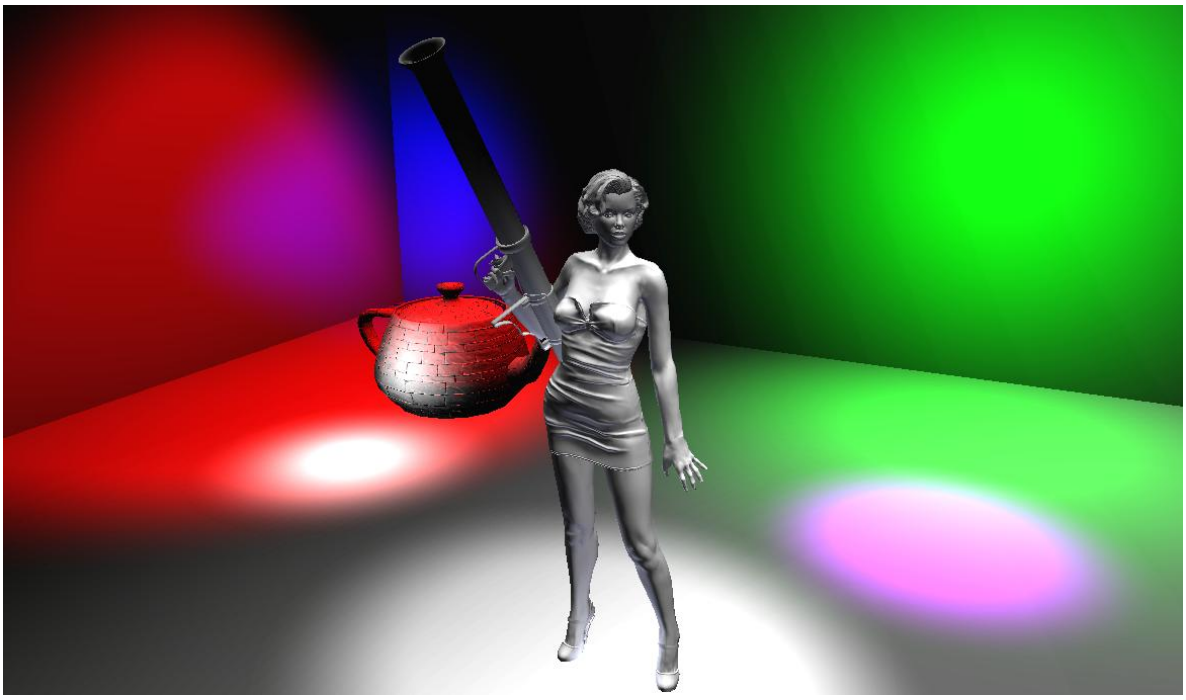


Ilustración 26. Escena final: Iluminación difusa

Vemos en la ilustración 26 cómo solamente con esta iluminación los volúmenes quedan completamente definidos. Llaman la atención especialmente las arrugas del vestido, con un alto nivel de detalle. La iluminación del foco que actúa sobre Marilyn se ha reforzado con una luz omnidireccional con poco radio debajo de ella, actuando como si fuera el rebote en el suelo de la luz del foco. Aparte de estas y la luz direccional base, el resto de luces omnidireccionales y de foco no tienen un propósito concreto y se han incluido con la finalidad de mostrar múltiples luces en la escena.

Podemos ver en este ejemplo que algunas de las luces aportan color directamente a la iluminación, independientemente de las texturas de los objetos.

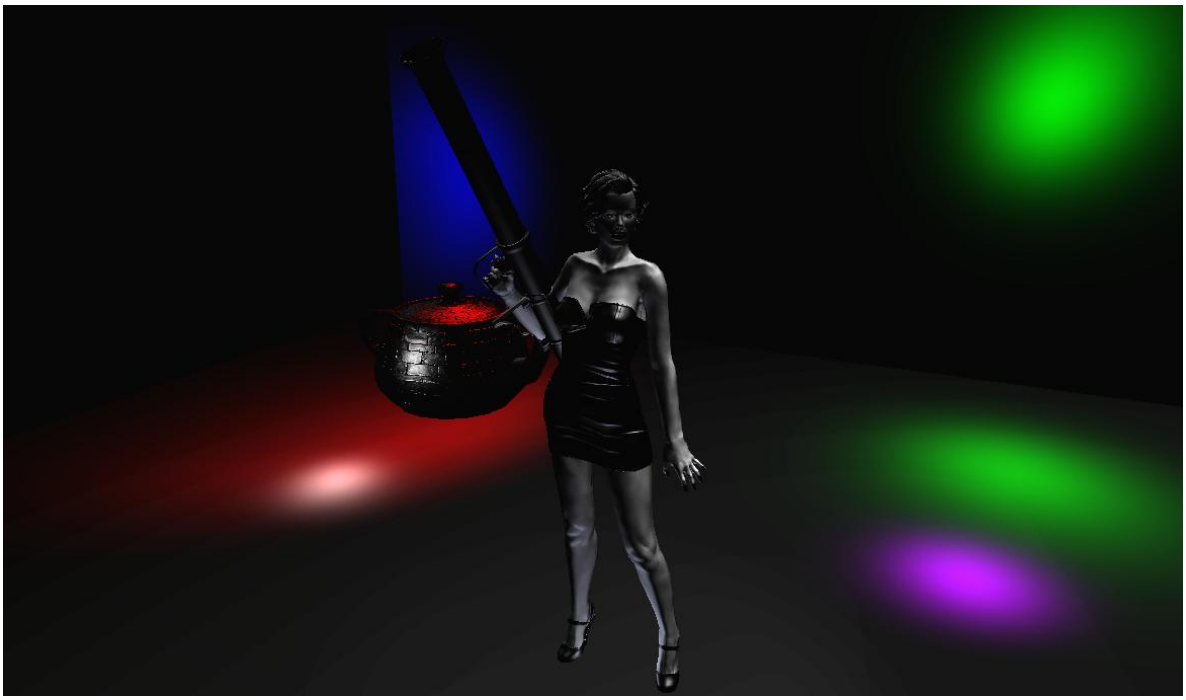


Ilustración 27. Escena final: Iluminación especular

Por último analizaremos la iluminación especular de la escena, visible en la ilustración 27. Hay que resaltar principalmente la poca iluminación que aporta, comparado con la ilustración anterior. Al igual que en esa, aquí la información de color existente corresponde únicamente a la luz, porque la textura de especular solo se ha tenido en cuenta para el coeficiente almacenado en su canal alpha.

El modelo de Marilyn nos sirve para distinguir distintos tipos de reflexión en un mismo modelo. El coeficiente utilizado para la piel hace que el brillo se extienda mucho más por la superficie, haciéndola más uniforme. En el vestido y los zapatos, en cambio, está mucho más concentrado ayudando a marcar algunas líneas. Aunque la piel aparece más iluminada, luego la intensidad del color de especular almacenada en el g-buffer (ilustración 25) la modulará reduciendo la cantidad de brillo.

En la última fase del proceso, el sistema combinará los g-buffers de color difuso y especular (ilustraciones 22 y 25) con las dos iluminaciones, difusa y especular (ilustraciones 26 y 27) y formará la imagen final (ilustración 21).

TRABAJOS FUTUROS Y CONCLUSIONES

CONCLUSIONES

Tras haber repasado el desarrollo del proyecto y haber observado los resultados obtenidos, es importante pararse brevemente a examinar cómo se adecúan a los objetivos planteados.

El objetivo más importante era conseguir renderizar escenas utilizando deferred shading sin que se pudiera apreciar visualmente un empeoramiento de la calidad gráfica. El resultado, como se ha visto en las escenas de prueba, es muy satisfactorio. La nueva arquitectura no tiene ningún defecto visual importante que pudiera condicionarnos a la hora de optar por una arquitectura u otra.

Aunque no se han efectuado pruebas exhaustivas de rendimiento, la respuesta obtenida en escenas medianamente complejas es muy positiva, manejando muchas luces sin mucho impacto. Recordemos que éste era uno de los aspectos más interesantes de esta nueva arquitectura, que permite una mayor flexibilidad en la iluminación al no limitar el número de luces. Escenas como la utilizada en el ejemplo final no habrían sido posibles de otra forma, ya que por ejemplo al objeto de la habitación le están afectando más luces de las permitidas con la implementación por defecto ofrecida por XNA.

También era muy deseable que el uso de la librería de renderizado fuera modular y de fácil uso. Gracias a haberse implementado como una librería separada del visor, no tiene dependencias innecesarias que compliquen el uso en otros proyectos. Por su parte, los procesadores de contenido encajan perfectamente dentro del pipeline de contenido de XNA y su utilización es tan simple como seleccionarlos de la lista de los procesadores de contenido cuando se compila desde el entorno de desarrollo, o especificarlo por parámetro si la compilación se hace como parte de algún proceso más complejo.

Existen, sin embargo, algunos puntos que deberían mejorarse antes de considerar esta arquitectura de render una alternativa adecuada al sistema tradicional. Por un lado, sería muy interesante poder mantener de una forma sencilla escenas con shaders distintos que interpretaran la iluminación de distintas formas para permitir técnicas avanzadas como subsurface scattering, efecto fresnel, etc. Lo más adecuado para esto sería convertir la arquitectura en un deferred lighting con una tercera etapa en la que se vuelva a procesar la geometría.

Por otro lado, ahora mismo la arquitectura no soporta ningún tipo de transparencia, ni siquiera alpha test. Para ser realmente efectiva, esta arquitectura debería manejar distintos tipos de transparencia sin limitar al desarrollador.

En general, podemos decir que, aunque aún no esté a la altura del completo sistema de renderizado tradicional, el resultado es muy positivo y demuestra que este tipo de arquitectura es perfectamente viable en motores más sencillos.

TRABAJOS FUTUROS

DEFERRED LIGHTING

La mejora más inmediata de nuestro sistema de deferred shading sería implementar la técnica de deferred lighting para facilitar el uso de diferentes tipos de materiales.

Se descartó como parte del proyecto original porque los materiales que se iban a usar eran muy básicos y era poco práctico el esfuerzo adicional para obtener los mismos resultados.

Si se quisiera implementar esta técnica sería necesario implementar una tercera pasada en la cual se renderizaría de nuevo la geometría usando esta vez shaders específicos de cada material.

Estos nuevos shaders se aprovecharían de los rendertargets de luz difusa y especular y utilizarían esa información junto con todos los parámetros adicionales que requiriese el material (parámetros que no se encontrarían entre los disponibles en los g-buffers) para calcular el color del pixel.

SOMBRAS

Un concepto que va siempre muy unido al de la iluminación (aunque las técnicas que lo cubren son muy diferentes) es el del dibujado de sombras. Cuanto más realista es la iluminación, más se echa en falta la presencia de las sombras.

Al separar el pintado de la geometría del de las luces, las técnicas de deferred rendering tienen ventajas para el dibujado de sombras. Aunque no todas las luces deben arrojar sombra por el coste que supone, aquellas que lo hagan restarán solamente el aporte exacto de iluminación que les corresponde, en lugar de estimar la cantidad de sombra que deben producir.

Las técnicas actuales de generación de sombras utilizan los llamados “shadow maps” para obtener sombras suaves, en lugar de las sombras duras obtenidas por las técnicas de “stencil shadows”. Ninguna de estas técnicas tiene ninguna incompatibilidad con las de deferred rendering.

Hay que tener en cuenta que no todos los tipos de luces pueden generar shadow maps. Las más adecuadas son las luces de foco, que pueden ser utilizadas sin ningún ajuste. Las luces omnidireccionales requieren 6 shadow maps distintos para generar sus sombras y por lo general se desaconseja su uso. Las luces direccionales por su parte pueden usar un shadow map, pero antes necesitan acotar el volumen al que afectarán, lo que determinará además la resolución disponible.

TRANSPARENCIAS

Como hemos indicado, uno de los grandes inconvenientes de usar este sistema es la imposibilidad de manejar superficies transparentes iluminadas. No obstante, muchos objetos transparentes no requieren iluminación, o puede usarse una iluminación aproximada. Es posible, por tanto, procesar en una pasada separada estos objetos.

Para implementar esta mejora es necesario identificar los materiales transparentes, para obviarlos en la primera pasada de generación de g-buffers. Estos mismos materiales deben ser pintados después de obtener la escena final, combinándose con el color del fondo según corresponda.

Si se implementa antes la técnica de deferred lighting, pueden combinarse los objetos transparentes en la tercera pasada si se ordenan adecuadamente (como ocurre en las arquitecturas de render que no son deferred).

SCREEN SPACE AMBIENT OCCLUSION (SSAO)

El modelo de luz ambiente que hemos presentado en este proyecto hemos visto que tiene un resultado completamente plano y uniforme. En la vida real la luz que se pretende modelizar con la luz ambiente, fruto de rebotes de otras luces, no llega igual a todos los lugares.

Existe una técnica muy utilizada en los juegos de última generación que estima la cantidad de luz indirecta que recibirá una superficie en función de los objetos que tiene en su entorno cercano. Cuanto más libre este un objeto, más cantidad de luz recibirá ya que no hay superficies que lo ocluyan. El cálculo se realiza en espacio de pantalla, es decir, solo sobre los pixeles visibles y está muy lejos de ser realista. El resultado obtenido, sin embargo, es relativamente creíble y añade volumen en aquellas zonas que no reciben otro tipo de iluminación.

Uno de los requisitos para esta técnica es la obtención de un buffer con la profundidad de los objetos. En nuestra arquitectura deferred este buffer ya está disponible puesto que es uno de los g-buffers utilizados para iluminar la escena, por lo que el coste de emplear esta técnica se reduce.

OTROS FILTROS

Del mismo modo que ocurre con el SSAO, existen muchas técnicas interesantes que se aplican en espacio de pantalla. El tener la escena previamente dividida en g-buffers facilita mucho el uso de este tipo de técnicas.

Por mencionar algunas, sería interesante implementar algún tipo de anti-aliasing, puesto que el deferred rendering hace imposible aprovecharse de las implementaciones tradicionales. El anti-aliasing morfológico MLAA y su variación el SMAA se aplican sobre el espacio de pantalla y serían una solución muy adecuada.

APENDICE I. GLOSARIO

Alpha: Canal adicional en una textura que guarda información en escala de grises, generalmente utilizado para indicar la cantidad de transparencia.

Binormal: Vector perpendicular al mismo tiempo al vector normal y al vector tangente de un vértice.

Deferred shading: Técnica de pintado separada en dos fases, una de geometría y otra de iluminación.

Deferred lighting: Modificación del deferred shading que añade una pasada más de geometría posterior para aquellos materiales que tengan formas distintas de calcular su resultado.

G-Buffer: Textura donde se almacena el valor para los pixeles en pantalla de un parámetro de la geometría de la escena.

Luz difusa: Tipo de luz reflejada por un material después de haber absorbido la correspondiente al color de éste.

Luz especular: Tipo de luz reflejada sin absorberse en una misma dirección.

Normal: Vector perpendicular a la superficie en un vértice.

Rasterizador: Parte de los mecanismos de renderizado que se encarga de trazar el polígono formado por varios vértices y determinar a qué pixeles afecta.

Renderizar: Proceso de mostrar gráficos 3D en pantalla.

Rendertarget: Textura que recibe el proceso de renderizado, generalmente para posteriormente mostrarse por pantalla, aunque puede ser utilizada simplemente como textura para otros efectos.

Shader: Programa personalizado que se ejecuta en la tarjeta gráfica.

Stencil: Buffer opcional de las dimensiones del rendertarget que almacena valores enteros para cálculos sobre el pintado de los pixeles.

Tangente: Vector perpendicular a la normal de un vértice.

APENDICE II. REFERENCIAS

[1] Wikipedia – Deferred shading. *Diciembre 2013*
http://en.wikipedia.org/wiki/Deferred_shading

[2] Tutorial deferred shading por Shawn Hargreaves. *Marzo 2004*
http://amd.wpengine.com/wordpress/media/2012/10/D3DTutorial_DeferredShading.pdf

[3] Deferred rendering en Killzone 2. *Julio 2007*
http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf

[4] XNA Developer center. *Diciembre 2013*
http://msdn.microsoft.com/en-us/aa937791.aspx

[5] Unity página principal. *Diciembre 2013*
http://unity3d.com/

[6] UDK página principal. *Diciembre 2013*
http://www.unrealengine.com/udk/

[7] Unreal Engine página principal. *Diciembre 2013*
http://www.unrealengine.com/